AFRL-IF-RS-TR-2004-102
Final Technical Report
April 2004

AN ANALYTICAL INVESTIGATION OF SOFTWARE MUTATION FOR INCREASED INFORMATION SURVIVABILITY

Reliable Software Technologies Corp.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F166

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-102 has been reviewed and is approved for publication

APPROVED: /s/
JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
APRIL 2004

3. REPORT TYPE AND DATES COVERED
FINAL **JUL 97 – JAN 00**

4. TITLE AND SUBTITLE

AN ANALYTICAL INVESTIGATION OF SOFTWARE MUTATION FOR
INCREASED INFORMATION SURVIVABILITY

5. FUNDING NUMBERS
C - F30602-97-C-0322
PE - 62301E
PR - F166
TA - 40
WU - 06

6. AUTHOR(S)
C. C. Michael, Aron Bartle, John Viega, Alexandre Hulot,
Natasha Jarymowycz, J. R. Mills, Brian Sohr, Brad Arkin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Reliable Software Technologies Corp.
21351 Ridgetop Circle, Suite 400
Dulles VA 20166

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2004-102

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: John C. Faust/IFGB/(315) 330-4544 John.Faust@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*
The work reported here has three major components. First, we report on the development of an analytic framework for judging the success of diversification schemes at retarding malicious attacks. We attempt to quantify the relationship between the amount of work needed by a defender to achieve a certain level of diversity, and relate it to the amount of work needed by an attacker to subvert the resulting system. Second, we report on a prototype system for automating the creation of diverse programs. This is, in fact, a general-purpose mutation system wherein the modification of software is controlled by mutation scripts or meta-programs that drive the mutation of source code. Finally, we report on a further prototype system for boosting information-system survivability by means of diversity. This system monitors the behavior of programs and ensures their compliance with semantic constraints describing their intended behavior, so that undesired behavior can be detected. Monitoring all aspects of a program's semantics would be infeasible, but in a diverse system, different replicas of a program can monitor different, randomly selected aspects of software behavior.

14. SUBJECT TERMS
Software Diversity, Software Mutation, Mutation Scripts, Semantic Constraints, Abstract Interpretation, Behavior Preserving Diversity

15. NUMBER OF PAGES 43

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

1 Introduction: The role of behavior-preserving diversity in retarding attacks on information systems

Redundancy and diversity are often used to increase reliability and survivability in critical systems. Redundant copies of crucial subsystems can serve as backups for one another, at least if they do not all fail at the same time. We might expect that redundancy and diversity could confer the same benefits on information systems facing malicious attacks, but this is not always the case. Redundant components serving the same purpose are often vulnerable to the same attacks, and, to make matters worse, these attacks can often be replayed at very little cost. As a result, the redundant components *can* all fail at the same time under a barrage of identical, repeated attacks.

A classic example of this phenomenon is the 1989 Internet worm (see [24]). This program exploited a vulnerability common to almost all of the systems that comprised the Internet at that time. The worm was self-replicating, so the cost of launching it against every system on the Internet was the same as the cost of launching it against a single system, at least in terms of human effort.

Viruses are another obvious example of an attack that is cheap to replay once it is developed, and viruses also require little human effort after the are developed. But even if an attack does not spread under its own power, it can still be repeated by encoding it as a script, since the replay of script requires little effort. Such scripts are often published, and although this allows information systems to be immunized against the attack, it also gives more people the ability to replay the attack. This also increases the number of supposedly redundant systems that can be undermined in a relatively short time.

One proposed solution is to make redundant systems diverse as well. If the replicas of a critical component were not vulnerable to the same attacks, we would expect redundancy to work as well in the face of malicious attacks as it does in the presence of other adverse circumstances. We would not expect redundancy to thwart an attack altogether, since that is not its purpose, but we would expect a redundant system to survive significantly longer than a system with no redundancy. The goal of this report is to quantify the benefits that can be obtained when redundancy and diversity are combined.

The work reported here has three major components. First, we report on the development of an analytic framework for judging the success of diversification schemes at retarding malicious attacks (Section 3). Here, we attempt to quantify the relationship between the amount of work needed by a defender to achieve a certain level of diversity, and relate it to the amount of work needed by an attacker to subvert the resulting system. Second, we report on a prototype system for automating the creation of diverse programs (Section 4). This is, in fact, a general-purpose mutation system wherein the modification of software is controlled by mutation scripts, or meta-programs that drive the mutation of source code. Finally, we report on a further prototype system for boosting information-system survivability by means of diversity (Section 5.1). This system monitors the behavior of programs and ensures their compliance with *semantic constraints* describing their intended behavior, so that undesired behavior can be detected. Monitoring all aspects of a program's semantics would be infeasible, but in a diverse system, different replicas of a program can monitor different, randomly selected aspects of software behavior.

Before presenting the detailed description of our work and results, we give a high-level summary in the next section.

2 Summary of Results

2.1 Summary of work performed

2.1.1 A framework for measuring the efficacy of software diversity in retarding malicious attacks

The first task of this effort was to construct a framework for quantifying the effectiveness of diversity in combating malicious attacks. In particular, the amount of work needed by the attacker to overcome a certain level of diversity was to be compared to the defender's effort in creating that diversity.

Since there are several sources of nondeterminism in this problem, such as the attacker's choice of attacks and the defender's choice of diverse systems, it seemed desirable to use a probabilistic framework. The most obvious approach would be to use an analogy with software testing. With such an analogy the attacker's goal (like that of a software tester) would be to break the program, and we could describe the attacker's behavior in probabilistic terms the same way test inputs are often described by statistical models (see [17, 19]). The problem with such an approach is that it may be impossible to describe an attacker's behavior with any useful probabilistic model.

However, it might be possible to model the defender's behavior with statistical methods. In particular, we can simply *ask* the defender to behave randomly in some respects, if such behavior enhances security. We found it useful to assume that the defender has some pool of potential replicas, and chooses one or more of these at random when creating a diverse system. Under this assumption, a very natural measure of diversity, at least with respect to a given attack, is the probability that a given replica will withstand the attack.

This measure of diversity is hard to put into numbers without knowing what attacks will be used and what replicas the defender has available. However, we can use this number as a variable, and ask: "if the potential replicas have a given level of diversity, how much does the defender gain from using redundant replicas?" Thus, we might create a curve showing the defender's work-factor on one axis and the attacker's work-factor on the other. The slope of the curve might depend on the (unknown) diversity factor, but we can still use the shape of the curve to determine what payoff redundancy will have for the attacker.

The model we developed allows us to describe a fairly rich variety of attack scenarios. Unfortunately, there are many scenarios where redundancy does not pay off satisfactorily. In these scenarios, the attacker's work factor only grows logarithmically compared to the defender's work-factor. This happens because of a shotgun effect that benefits the attacker: an attack intended to defeat one replica may actually defeat many. Although there are many cases where this does *not* happen, especially the case where the attacker has limited time or a limited pool of attacks, this result raises a question about the ultimate utility of diverse systems.

2.1.2 A general-purpose, configurable software mutation system

To investigate the practical aspects of software mutation as a means of information system diversity, we constructed a general purpose mutation system that works by modifying parse trees. The system is driven by mutation scripts that specify how the program is to be changed. Such a system has several potential applications in information system security:

- It allows generalized software patching. Since our system recognizes features of the parse tree, and not features of the source code as standard software-patch programs do, a single patch script can be applied to many different programs. For example, we wrote scripts that insert buffer-overflow guards on all local stacks in a program, and ones that prevent programs from using the stack altogether by changing stack-allocated variables to heap-allocated variables.

- Mutation scripts can be used to insert code that monitors program data. We previously used software instrumentation to insert checks for violations of security policy into programs, but that system was not general-purpose. Program behavior can also be monitored for purposes such as intrusion detection and program state inference.

- The intended role of the system was as a tool to boost information system diversity by randomly selecting and applying mutations that preserve the intended semantics of the program.

Unfortunately, we found the programming of mutation scripts to be clumsy work, for reasons discussed in Section 4.2. It also seems as though completely random software mutation — even if the mutants are semantically equivalent to the original code — is not a good approach to software diversification; the results described in Section 2.1.1 are only one reason for coming to this conclusion. However, part of our proposed work involved the extraction of semantic information from programs in order to determine when mutants preserved the semantics of a program. We shifted our focus by placing more emphasis on the use of semantic information, and thus exploring new, semantically based approaches to creating software diversity (as opposed to the syntactic approach of software mutation.) This is described in the next section.

2.1.3 A system for automatically extracting behavioral constraints from programs

Originally, we proposed to use semantic analysis as a way to determine if a mutant was semantically equivalent to the program being mutated. This was meant to ensure that the mutated program would have the same functionality as the original — the intent was to alter *unintentional* (or nonessential) semantics of a program, not to prevent it from fulfilling its intended purpose. However, in light of the apparently limited success of software mutation, we were also in search of improved methods through which information systems could be made diverse.

The methods we had intended to use to determine semantic equivalence, taken from existing work in software mutation testing, proved inadequate for our purposes because there were too many heuristic elements. Instead, we developed a new technique for representing program semantics, based on set constraints [13], constraints that specify execution paths, and a technique we developed for representing loops inductively. We developed software for extracting such semantic information from C/C++ programs.

We found that such semantic constraints could be used to monitor the integrity of executing software. Indeed, the kinds of attacks we had hoped to delay by means of software mutation could apparently be detected as deviations from the semantics we automatically extracted from source code. For example, viruses and buffer-overwrite attacks change the behavior of an executing program, and this change in behavior can be detected through the program's failure to obey the constraints we extract. We constructed a prototype that uses semantic constraints to detect deviant software behavior.

Two things differentiate our work from other existing systems for the extraction of software semantics, aside from the fact that our system is meant to enhance information system security, which others are not. First, our system is meant for use with procedural or object-oriented programs, while most existing systems are meant to be used with functional or logic programs. Secondly (and more importantly) our system explicitly recognizes that the extraction of semantic information is a difficult problem, not only because some program constructs are hard to handle but because the number of constraints (hence the resource consumption of a constraint-based technique) often grows exponentially in the size of the program being analyzed. Our system takes these issues into account, and can still be used when only incomplete semantic information is available. Specifically, we can simply monitor the program's compliance with a subset of the available constraints.

Here, diversity once again comes into play: if different constraints can be monitored independently (our experience with the prototype suggests that they can) then there is diversity among the constraints monitored within a particular program. Different constraint-sets can also be monitored in different copies of a redundant program.

Furthermore, it seems possible to *recover* from deviant semantic behaviors, since the constraints not only tell us when a program has deviated from its intended behavior, but also what the intended behavior was. In principle, the program could be forced back into a correct state, although we have not implemented this in our prototype.

Interestingly, our constraint based-approach can apparently be used to emulate the software mutation technique we originally planned (though we were not able to implement this extension under the existing contract). We can mutate the software constraints, instead of the original program, and if we do so, the constraint-monitoring system can tell us what each mutant *would have done.* This gives us the same benefits of an actual mutation system: we can detect when the mutants behave differently, and force the program into the state determined by a particular mutant. Furthermore, since our constraint monitoring system can monitor many constraints at once for a single program, we have an inexpensive way to maintain several mutated programs running in parallel. Instead of running all of the redundant programs individually, which requires special software or hardware to start the programs simultaneously and resolve differences in their behaviors, we can simply calculate how the redundant programs would have behaved if they really had been running in tandem.

3 Details of the Analytic Framework

3.1 Introduction

3.1.1 Diversity in software engineering

Redundancy is often used to increase the robustness of information systems. A distributed system like the Internet can continue to perform many functions, such as the routing of information, even when some subsystems fail.

A more ambitious goal is to stop failures from happening altogether by having the redundant subsystems perform the same functions at the same time. In this way, the desired function is still accomplished even if some subsystems fail. Here, it is pointless to have many copies of a single program running at the same time. Since software failures are caused by design flaws rather than aging or physical stress, all copies of a program would replicate its design flaws, and since all copies are processing the same data, they would all fail at the same time.

The proposed solution to this problem was to create a number of *different* programs from the same specification. Such programs could be developed by different programming teams working independently, and the hope was that this would result in redundant programs that failed independently of one another. Unfortunately, it was found [16] independently developed versions of a program did *not* always fail independently; the independent programming teams sometimes make the same design mistakes. This makes it difficult to determine the return on investment obtained from software diversification.

We can expect the same problem to be encountered when redundant copies of a program are subjected to malicious attacks. Since it is cheap to replay attacks, making many copies of a single program would not be cost-effective because they would quickly succumb to a single, repeated attack. On the other hand, the redundant components may be vulnerable to the same attacks even if the defender *tried* to make them diverse. In other words, the attempt to create diverse replicas may fail.

To make matters worse, some attackers may have better strategies than others. A skilled attacker may apply techniques that are broadly effective overall, and thus subvert many replicas of a component. There may also be an attack *paradigm* that is easy to apply to many replicas even if the attacks themselves only subvert one replica at a time. Thus, a good attack strategy can also cancel the benefits of software diversity.

3.1.2 Average-case behavior and randomized diversity

In analyzing the survivability of a redundant system, we do not want to deal with the worst possible case, because the worst cases are unrealistically bad. For example, the attacker might get lucky and destroy all components of a redundant system with a handful of randomly selected attacks. The attacker might also know the precise semantics of each component, and by sheer luck, happen to know an attack technique will subvert each one. We would rather perform an average-case analysis than spend time and effort describing situations that we hope will rarely if ever actually occur.

Thus, a statistical analysis suggests itself, all the more so since statistical approaches are so common in other areas of software assurance. For example, software reliability analysis assumes that users and the state of nature create certain conditions that may or may not make a program fail. If usage patterns and environmental conditions are treated as random phenomena, one can quantify the program's quality in terms of a *probability* of failure. One could argue that a malicious attacker is like the user or the environment, supplying inputs that may or may not lead to a breach of security (these are the attacks). One could then say that an information system has a certain probability of being subverted by a given attack (analogous to the probability of failure) and a certain probability of surviving a string of attacks (analogous to software reliability).

Unfortunately, an attacker's behavior may not be amenable to statistical analysis, since the attacker intelligently seeks out a program's weaknesses and focuses on them. We expect the attacker's behavior to be the result of conscious decisions in a constantly evolving strategy, so that future behavior cannot be predicted by observing the past as is done in statistical prediction. Although one empirical analysis, [15], found attacker behavior to possess properties that would make it statistically predictable, there is too little information at present to safely conclude that this is a general rule, especially since intuition suggests the opposite conclusion.

Speaking in more general terms, we expect that attacks will not be chosen at random, but rather on the basis of what seems most likely to succeed. Therefore, the choice will depend on the attacker's talent and general knowledge, on what the attacker has observed about the target system, what the attacker has learned during previous attack attempts, and also on the general techniques and paradigms being used by malicious attackers at the time in question. In short, past attacker behavior may not be useful in predicting future attacker behavior, because to do so we must take into account an enormous number of factors and contingencies that are difficult or impossible to measure.

This puts us in a difficult situation. On one hand, statistical approaches rely on past behavior to predict future behavior when it comes to actually predicting what will happen in a particular situation. On the other hand, if we want some sort of non-statistical assurance about the robustness of a redundant system, we have to construct a system that is immune to malicious attacks, and then prove it to be immune. Unless we are dealing with some pre-defined class of attacks and a relatively simple system, this can be difficult or impossible. All of this suggests that we cannot confidently make predictions about the behavior of a system that is being maliciously attacked.

However, we can still use an analytical framework to determine what factors influence the success or failure of a redundancy in a system that is being maliciously attacked, and to say something about how important each of these factors is. That is what we hope to do in this paper.

In this paper, our approach is to ask the *defender* to adopt a randomized strategy. This approach, which has been used in other areas to facilitate the average-case analysis of algorithms, can also be used for that purpose when we analyze the effects of diversity. The idea is that the defender has a pool of *potential* replicas for each redundant component. This pool of replicas could actually be a database of software modules, or it could be an abstract concept that arises when the defender uses a randomized process to create new replicas.

3.2 Framework

In this paper, assume that there are a number redundant, randomly selected subsystems that are under attack. In principle, we want to know how many attacks will be needed to subvert all of the subsystems, but since the subsystems are selected at random, we cannot know this with certainty. Instead, we have a certain confidence that some subsystems will still survive after a certain number of attacks, or, conversely, a certain confidence that all the subsystems will be destroyed. A confidence of either sort can be used to quantify the effectiveness of the diversification scheme.

We find it useful to examine the problem at three levels of detail. First, we ask how likely it is that a randomly selected subsystem will survive a given attack. This is the basic measure of how effective the diversification scheme is: if the defender finds it hard to create a system that is immune to a certain attack, then most subsystems will succumb, so that the probability of survival will be low.

Next, we ask how likely it is that a single subsystem will survive a series of essentially simultaneous attacks. We call such a series of attacks a *salvo*. In this context, "essentially simultaneous" means that anything the attacker learns from the outcomes of attacks cannot be applied in other attacks of the same salvo. Likewise, if the defender has an intrusion response system, it cannot learn to recognize new attacks soon enough to use that knowledge during the same salvo.

Finally, we consider the case where there are several redundant subsystems. We will ask how likely it is that at least one subsystem will survive a series of attack salvos.

3.3 The probability that a randomly selected subsystem will survive one attack.

We begin by considering the effects of a single attack, which we regard as being fixed. In other words, it is as if an attack has already been chosen, and the defender is charged with finding a subsystem that will withstand the attack. This is done by selecting one of a pool of redundant, diverse systems, and the defender's chance of success is the probability of selecting a system that will withstand the attack.

The probability that a randomly selected system will survive a malicious attack is, in some sense, the basic measure of the diversification strategy's success. For example, the supposedly diverse variants of the

Unix finger daemon may appear to use completely different implementations, but the developers may not have known that some attacks exploit vulnerabilities in the C-language library. If each of the diverse copies of the finger daemon use this library, they will all be vulnerable to this attack. If the attacker uses that particular attack, the probability of selecting a daemon that survives the attack is zero; the attack destroys all of them.

In some sense, this survival probability quantifies the diversity of the redundant systems. If the redundant systems fail to be diverse in the sense that they all succumb to the same attacks, the survival probability will tend to be low. On the other hand, if the redundant systems are never vulnerable to any of the same attacks, the survival probability will be high. As an illustration, if the defender has 100 systems, no two of which are vulnerable to any single attack, then the probability of selecting a system that survives a given attack is 0.99.

Given that this basic survival probability reflects the diversity of the redundant subsystems, it would be intellectually appealing to express the survival probability in terms of some natural measure of diversity. Unfortunately, it is another matter to ensure that this measure of diversity is actually useful. It seems that the defender's most natural question is "how likely is it that the next replica I generate will be survive the attacker's next attack?" The author has not been able to find another basic measure of diversity that reflects the practical problems of software diversification as well as this one. Therefore, we use the survival probability as the basic measure of success for a diversification scheme, with respect to a single attack.

3.3.1 Example: choosing a secure server

The purpose of this paper is to say what factors affect the success or failure of a diversification scheme, rather than to come up with exact predictions of what will happen when diversification is applied to a real system. However, a concrete example will illustrate the concepts underlying our framework.

Suppose that some critical operation is to be carried out in the presence of malicious attacks. Redundancy is achieved by making several sites available for carrying out the operation, and the defender randomly selects the site that is to be used. The attacker will attempt to disrupt the critical operation. The attacker does not know what site the defender will use, but this, in itself, makes no difference, because any attack can easily be launched against every possible site.

Currently, we are concerned with what will happen if the attacker chooses some particular attack; suppose that a particular Windows NT denial of service attack will be used. Of the sites available to the defender, ten use Unix systems and are therefore immune to the attack. The other 21 sites use NT, but twelve of these sites have installed patches that prevent the attack in question. Therefore, if the defender chooses a site uniformly at random, the probability of surviving this particular attack is 22/31.

Note that the defender could simply choose one of the immune sites if it were actually known that this particular denial-of-service attack would take place. However, we do not assume that this attack will actually be used, much less that the defender actually *knows* it will be used. At present, we are only asking what will happen *if* the attack takes place.

3.3.2 How easy is it to create diversity?

In an earlier example, we assume that a defender had 100 redundant replicas of a system, and that no two of these were vulnerable to the same attack. In that case, it is easy to see that the survival probability with respect to *any* attack is at least 99%; no matter what attack is used, the defender has a 99% chance of selecting a system that is immune to it.

However, it may be a somewhat utopian world in which no two replicas are vulnerable to the same attack. The more replicas are added to the pool, the more difficult it becomes for the defender to ensure that this property continues to hold. When the the second replica is added, the defender only has to ensure that it doesn't share any vulnerabilities with the first replica, but when the thousandth replica is added, the defender has to make sure that it has none of the vulnerabilities of the other 999 replicas. Thus, the defender's job seems to become more difficult as more replicas are added.

It is more realistic to assume that replicas are created according to some process that creates a certain level of immunity to attacks. In the example of the last section, we might say that there was a certain chance of choosing Unix as the operating system at a given site, and if Unix were not chosen, that there was still a certain probability of installing the patch that prevents the denial-of-service attack. Thus, a system emerging from this "process" has about a 22/31 probability of being immune to the given attack.

In this situation, the *number* of redundant replicas does not have a great effect on diversity. If one million replicas are created by the same process, and that process has a 70% chance of creating a system immune to a certain attack, then the survival probability will be about 70% regardless of how many replicas there are. Therefore, it is the quality of the replication process, rather than the number of replicas, that determines the effective diversity of a redundant system. We will find this to be a continuing theme: under plausible assumptions, the sheer *number* of diverse replicas does not greatly affect the success of a diversification scheme.

3.4 The likelihood that an attack salvo will subvert a single server.

Once we have survival probabilities for attacks, we can easily bound the probability that an attack salvo will subvert a server. Although we are still only considering the behavior of a single replica, we now find it useful to account for the properties of the entire pool of replicas. For a fixed attack, we select n replicas and define a *behavior string* for that attack, with respect to the replicas we have chosen. The behavior string is just a string of ones and zeros; the ith element of the behavior string is 1 if the attack subverts the ith replica, and zero otherwise. Clearly, if two attacks behave in exactly the same way for *all* replicas, then the survival probability is the same for both attacks.

Given an attack salvo a, Let $B(a) = \{b_1, b_2, \cdots, b_n\}$ be the *set* of behavior strings associated with the attacks in a salvo. For a given behavior string, we can define $\epsilon(b, r)$ as the survival probability with respect to b, given that the replica is selected at random from the set r (For the moment, r can simply be considered to be the set of all available replicas, but later in the paper, we will find it useful to explicitly consider replicas that are selected from certain subsets of this set.) Let $D(b_1), D(b_2), \cdots, D(b_n)$ be the events in which the randomly selected replica is subverted by an attack with behavior string $b_1, b_2, \cdots b_n$, respectively, then the probability that one or more attacks subvert the replica is no larger than $\sum_{b\in B(a)} \mathbf{P}(D(b))$. (It may be less if there are dependencies between behavior of one attack and that of another.) But $\mathbf{P}(D(b_i))$ is just one minus the survival probability with respect to attacks having the behavior string b; it is $1-\epsilon(b,r)$. Thus, the probability of a randomly selected replica being subverted by the salvo is at most $\sum_{b\in B(a)}(1-\epsilon(b,r))$. If we have a lower bound ϵ_0 on $\epsilon(b,r)$ for all behavior strings b, then the probability of a replica being subverted is at most $|B(a)|(1-\epsilon_0)$, where $|B(a)|$ denotes the size of the set $B(a)$.

This quantity can be greater than one, since it is only an upper bound on a probability rather than being a probability in itself (when the bound is greater than one, it means that nothing useful came out of this bounding method, since it only tells us what we already knew, namely that the probability of the replica being subverted is less than or equal to one). We would like to use a bound that explicitly takes into account the fact that the replica's probability of being subverted is never greater than one. Thus, if R is a randomly selected replica and $a = \{a_1, a_2, \cdots, a_n\}$ is a set of attacks, we define

$$F(a, r) \stackrel{\text{def}}{=} \begin{cases} \sum_{b\in B(a)}(1-\epsilon(b,r)), & \text{if } \sum_{b\in B(a)}(1-\epsilon(b,r)) < 1; \\ 1, & \text{otherwise.} \end{cases}$$

Often the set of available replicas has some structure that affects the likelihood of an attack being successful. For example, replicas that have been patched in order to repair a specific vulnerability will categorically have a different survival probability from replicas that have not been patched, and replicas implemented in different operating systems will have fundamentally different behaviors when they are attacked. This makes it useful to partition the set of possible replicas, and perform an analysis like the one above separately for each element of the partition. If we let $r_1, r_2, \cdots, r_n$ denote the elements of such a partition, then the probability of subverting a randomly selected replica is no greater than

$$\sum_i \sum_{b\in B(a,r_i)} (D(b) \mid r_i)\,\mathbf{P}(r_i) = \sum_i \sum_{b\in B(a,r_i)} (1-\epsilon(b,r_i))\mathbf{P}(r_i)\,,$$

where $(D(b) \mid r_i)$ denotes the probability that a randomly selected replica will be subverted by b, given that the replica is selected from the subset r_i, and $\mathbf{P}(r_i)$ is the probability of selecting a replica from the subset r_i.

3.4.1 Example: An artificially large number of attacks

In the example of Section 3.3.1, suppose that the attacker has several attacks available, 30 of which affect NT systems, but one of which is effective against Unix. Nine of the defender's ten available Unix platforms have been patched and are not vulnerable to the attack. In this situation, it makes sense to partition the platforms into those running Unix (call this set U) and those running NT (we call this set NT).

The probability of selecting a platform that will be subverted is less than

$$\mathbf{P}\left(\bigcup_{b \in B(a,U)} D(b,U)\right)\mathbf{P}(U) + \mathbf{P}\left(\bigcup_{b \in B(a,\mathrm{NT})} D(b,\mathrm{NT})\right)\mathbf{P}(\mathrm{NT}) \leq F(a,U)\mathbf{P}(U) + F(a,\mathrm{NT})\mathbf{P}(\mathrm{NT})\,.$$

Instead of selecting a platform *uniformly* at random, the defender may favor Unix systems, suspecting that they are vulnerable to fewer attacks. Specifically, suppose $\mathbf{P}(U) = 0.7$ and $\mathbf{P}(\mathrm{NT}) = 0.3$, so that the probability of a successful attack is $0.7F(a,U) + 0.3F(a,\mathrm{NT})$. Note that all the NT attacks have the same behavior string with respect to the Unix platforms; in all cases it is a string of ten zeros. The Unix attack has its own behavior string, which contains nine zeros and a one. Thus, $B(a,U)$ has two elements. Furthermore, the survival probability for one of these two behavior strings is 1, since it represents the NT attacks. If the survival probability for the other attack is 0.9 when a Unix-based replica is selected, then $F(a,U)$ is at most $(1 - 0.9) = 0.1$.

On the other hand, suppose there are 30 NT attacks, and that each NT attack has its own behavior string with respect to the NT platforms, so that $B(a,\mathrm{NT})$ has at most 31 elements. (The Unix attack also has a behavior string of all zeros with respect to the NT platforms, but that may or may not be the same as one of the behavior strings derived from the NT attacks.) As above, suppose that the survival probability among NT platforms is 4/7 for each NT attack. Then $F(a,\mathrm{NT})$ is at most $|B(a,\mathrm{NT})|(1 - 0.57)$, but since this is greater then 1, $F(a,\mathrm{NT})$ is just 1. Thus the probability of selecting a server that will be subverted is less than $0.7F(a,U) + 0.3F(a,\mathrm{NT}) = 0.07 + 0.3 = 0.37$.

Suppose the attacker increases the number of NT attacks to 100, but suppose that the survival probability among NT attacks is still 0.57. The increased number of attacks may (or may not) make $B(a,\mathrm{NT})$ larger, but since the attacks are only effective against NT platforms, this can only make a limited contribution to the probability of subverting a randomly selected replica. In fact, $F(a,\mathrm{NT})$ was already as large as possible before the new attacks were added, so our previous bound of 0.37 is unaffected by presence of the additional attacks.

3.5 One or more salvos against many replicas

Our main interest is in redundant systems, that is, systems containing many replicas of critical subsystems. In this section we develop a framework for dealing with such systems. Instead of selecting a single replica at random, the defender now selects n such replicas. Our description of the attacker's strategy is once again based on behavior strings, but we are now interested in behavior strings on randomly selected sets of replicas. Just as we defined the behavior string of an attack to be a string containing a 1 or a 0 for each replica, so we can define the behavior string of an attack salvo. The string contains a 1 in the ith position if the *salvo* subverts the ith replica, and otherwise the string contains a 0. If s is an attack salvo and r is a set of replicas, we use $B(s,r)$ to denote the set of behavior strings obtained if the attacks in s are applied to the replicas in r.

If the defender randomly selects a set of n replicas R^n (e.g., R^n is a random variable), we are interested in the expected value of $|B(s,R^n)|$, (that is, the expected number of behavior strings that can be observed when the attacks are used against the replicas selected by the defender.)

This quantity, $\mathbf{E}(|B(s,R^n)|)$, is commonly used in machine learning to describe the diversity of hypotheses generated by a learning algorithm. Here, we use it to describe the diversity of the set s of attack salvos.

Our result for this section follows directly from a similar one that is well-known in machine learning [4]. If the survival probability of a single replica against an attack salvo (not just a single attack) is greater than η for all salvos, and the defender selects a set of n replicas, then the probability that all replicas will be subverted is less than

$$2\mathbf{E}\left(|B(s, R^{2n})|\right) 2^{-n\eta/2} \tag{1}$$

(Note that R is raised to $2n$ instead of n; this results from a proof detail that is beyond the scope of this report. The underlying proof is due to [4] and can be found in that paper). The physical interpretation of this bound depends a great deal on the contents of an attack salvo. Indeed, this flexibility is the motivation for expressing this result in terms of attack salvos when it could just as easily have been expressed in terms of individual attacks. We illustrate the application of this bound with some examples.

We can use this inequality to find a lower bound-size of the attack salvo needed to subvert a redundant system, if we have an *upper* bound on $B(s, R^{2n})$. But note that it is only a lower bound. In the following examples we will see cases where it is not positive. Since we already know that one or more attacks are needed to subvert any system, a lower bound of, say, -7, is trivial and fails to tell us anything new. When this happens, it means we have encountered a situation where the current framework is not helpful.

3.5.1 Example 1: Attacks that can be replayed cheaply

Consider a situation where new attacks are developed over time, but existing attacks can be replayed cheaply. This reflects what often happens in reality; existing attacks are automated so that they can be replayed instantly, but new attacks are developed at a comparatively slow rate. Often, the development of a new attack takes place only after the accidental discovery of a previously unknown vulnerability.

We will model this situation by starting a new salvo each time a new attack is discovered. Because of the comparatively slow pace at which new attacks tend to appear, we will say that the system recovers fully between salvos. In other words, there are no cumulative effects from one salvo to the next, so we only apply (1) to one salvo at a time, and if we want to see what happens over time, we simply apply (1) to the first salvo, then to the second salvo, and so on.

Since attacks can be cheaply replayed, each salvo contains the newly developed attack as well as all the old attacks. Thus, each salvo contains one more attack than the previous one, and we ask how large the salvo has to get before there is a high likelihood of subverting the entire system.

However, in this example we will study the effects of redundancy in isolation. We will say that the survival probability with respect to a single attack remains constant. Thus, the defender does not have an intrusion response mechanism that prevents old attacks from being reused, since this would cause the survival probability with respect to those attacks to go up over time. It also means that overall, each attack has about the same effectiveness, which implies some minimum level of competency on the part of the attacker.

Since we are only considering one salvo at a time, $|B(s, R^n)|$ is 1. We get η from the results of Section 3.4; if the survival probability (with respect to one attack) is always at least ϵ, then the probability that one randomly selected replica will be destroyed by an attack salvo is less than $k(1-\epsilon)$, where k is the number of attacks in the salvo. Thus the survival probability with respect to a single *salvo* containing k attacks is at least $1 - k(1-\epsilon)$.

In this example, the bound (1) simply becomes

$$2^{1-n(1-k(1-\epsilon))/2}, \tag{2}$$

where n is the number of replicas. Since this is an upper bound on the probability that all replicas are defeated, the attacker would like it to be large. For example, if the attacker wants to have a 99% chance of subverting all replicas, it is necessary to make (2) larger than 0.99 through an appropriate choice of k. To determine how many attacks the salvo must contain, assume that the attacker's goal is to ensure that (1) is at least as large as some constant δ. Thus, the attacker wants to choose a k such that $2^{1-n(1-k(1-\epsilon))/2} \geq \delta$. Solving this inequality for k, we see that the attacker should use at least

$$\frac{1 + 2/n \log(\delta/2)}{1-\epsilon}$$

attacks. In fact, the number of attacks is less than $1/(1-\epsilon)$ regardless of the number of replicas or the choice of δ. Thus, as far as this example is concerned, adding more replicas does not guarantee greater survivability. Only if the the defender increases ϵ, the survival probability of a single replica with respect to a single attack, does the survivability of the redundant system go up. But increasing ϵ would also increase the survivability of a non-redundant system.

Since (1) is only an upper bound, using $1/(1-\epsilon)$ attacks is a necessary condition for subverting all replicas. It is not a sufficient condition, and in that sense, we have only failed to demonstrate that adding more replicas increases the survivability of a redundant system. But it straightforward to show that the *necessary* number of attacks only grows logarithmically in the number of replicas when the survival probability is high.

3.5.2 Example 2: Intrusion response by means of patches

Suppose we are in the same situation as in the previous example, except that the long period of time that elapses between the development of new attacks allows the defender to patch up vulnerabilities to the old attacks. Thus, an attack salvo doesn't contain all past attacks (or if it does, those attacks do not effect the behavior strings or the survival probability, so that the effect is the same as having only one attack per salvo).

In the previous example, anything that happened during one salvo happened again in all subsequent salvos, since all old attacks were replayed in each salvo. Thus, we did not lose any information by considering only the last salvo. In this case, we have to consider all salvos because all of them are different, and each salvo gives the attacker one chance to subvert the system.

As before, let us assume that we have a lower bound on the survival probability that is valid for all attacks. In this case, this is also a lower bound on the probability that one replica will survive an entire salvo, since there is only one attack per salvo. On the other hand, the need to consider all salvos means that $B|(a, R^n)|$ may be as large as the total number of attacks. Thus, if there have been k attacks and there are n replicas, (1) becomes $2k2^{-n\epsilon/2}$. Solving for k as above, we see that the attacker needs $\delta 2^{n\epsilon/2-1}$ attacks in order to have confidence δ that all replicas will be defeated.

Here, the situation is completely different from the last example. The number of attacks needed to subvert the system grows exponentially both in the number of replicas and in the survival probability. The difference between this example and the last is that a single attack has to subvert all replicas. If the diversity of the replicas is moderately large (e.g., the survival probability is moderately high) and the number of replicas is large, then the likelihood of subverting all replicas with one attack quickly becomes miniscule. The attacker must continue to launch attacks until the defender gets unlucky, in spite of considerable odds in the defender's favor.

3.5.3 Example 3: Delayed intrusion response

In the previous section, we described an intrusion response technique that is common but not usually associated with the term "intrusion response." Usually, intrusion response systems are assumed to detect attempted intrusions automatically. Often, such systems are adaptive in that they learn to recognize new attacks, but the system may have to see the attack more than once to do this.

To model such a situation (albeit somewhat crudely) we assume that an attack can be replayed k times before the defender is able to prevent it from being effective. As in the previous, we model this situation by using several salvos, because we need to account for the possibility that the attacker will get lucky at any time during the attack process. Each salvo will contain a newly developed attack, but each salvo will also contain $k-1$ old attacks that are being replayed.

The probability of a single, randomly selected replica being subverted by a single salvo is at most $k(1-\epsilon)$. As in the previous example, the number of behavior strings is no greater than the total number of salvos, which, in this case, is the same as the number of new attacks that the attacker devises. Thus, (1) becomes

$$2\mathrm{k}2^{-n(1\ -k(1-\epsilon))/2}.$$

If we once again solve for as above, we see that

$$\delta 2^{n(1-k(1-\epsilon))/2-1}$$

attacks are needed for the attacker to have confidence δ that all replicas will be subverted. Thus, as long as k is not two large, the number of necessary attacks grows exponentially in the number of replicas.

(If k is large, the number of necessary attacks can be less than one according to this result. We know that at least one attack is necessary to subvert the system, so in that case the bound becomes trivial and tells us nothing that we did not know already. That means that our framework was not able to provide meaningful lower bounds on the number of attacks needed to subvert the system.)

3.5.4 Example 4: A fixed number of attacks

For this example, suppose that the attacker uses an attack script, but does not have enough skill to develop new attacks. We might expect that the defender, having seen the attack script as well, might have patched the vulnerabilities that the script exploits, in which case the attack simply fails outright. However, for this example, we assume that the defender has not systematically applied patches.

Since the attacker cannot change the likelihood of a successful attack, its success or failure depends only on the defender. Therefore, we ask how many replicas the defender needs to be confident of surviving the attack.

As before, we assume that the probability of a single replica being subverted is at most $k(1-\epsilon)$, where ϵ is a lower bound on the survival probability against any individual attack in the script, and k is the number of attacks in the script. The attacker has only one salvo to launch, so (1) becomes $2^{1-n(1-k(1-\epsilon))/2}$. This is the same bound as in example 1, but here the attacker is unable to take advantage of that fact by increasing k.

However, by comparing this example to example 1, we see that we cannot guarantee any benefit from simply adding more replicas. Instead, it is necessary to increase the survival probability. Thus, the process by which replicas are created must be improved, if possible.

(Of course, this can often be achieved if the defender knows the attack script. In that case, the necessary vulnerabilities can hopefully be patched. If they can, the survival probability becomes 1, and as result the value of k needed to subvert the system becomes infinite. It essentially becomes impossible for the attacker to succeed.)

3.5.5 Example 5: Hyperdiversity.

Suppose the defender's replicas can be partitioned into sets $r_1, r_2, \cdots, r_n$, in such a way that the members of one subset are not vulnerable to any of the same attacks as the members of any other subset. (This might happen if the replicas in different subsets employ different operating systems.) Let $1 - (D(b, R) \mid R \in r_i)$ denote the survival probability given that a replica is selected from the subset r_i. If we leave this probability constant but increase the number of subsets, then the overall probability of a randomly selected replica being subverted, namely

$$\sum_{i=1}^{n} (D(b, R) \mid R \in r_i) \, \mathbf{P}(R \in r_i) \,,$$

will stay the same if the defender has the same probability of selecting a replica from any of the subsets (e.g., $\mathbf{P}(R \in r_i) = 1/n$ for all i). We do, indeed, want the survival probability to remain constant when we increase n, because we want to study the effects of increasing n (e.g., increasing the number of different operating systems on which a replica might run) in isolation, not in conjunction with an overall increase of the survival probability. Therefore, we assume that the defender does, in fact, have the same probability of selecting a replica from any of the partitions. Once again we also assume that we have a lower bound, ϵ in the survival probability, so that $\mathbf{P}(D(b, R)) \, R \in r_i \leq (1-\epsilon)$ for all i and all b. (Note that unless ϵ is zero or one, there must be at least two replicas within each partition for this to be possible.)

As before, we assume that attacks cost nothing to replay, so we only consider the effects of a single salvo. Let a_i denote the attacks in this salvo that are potentially effective against replicas from the set r_i (e.g., they work against replicas running under the particular operating system that r_i represents.) The probability of subverting a randomly selected replica is no greater than

$$\sum_{i} \sum_{b \in B(a_i, r_i)} (D(b, R) \mid R \in r_i) \, \mathbf{P}(R \in r_i) \,. \tag{3}$$

We have assumed that the defender selects replicas so that $\mathbf{P}(r_i) = 1/n$ for all i, so (3) is just

$$\frac{1}{n}\sum_i \sum_{b \in B(a_i, r_i)} (D(b, R) \mid R \in r_i) \leq \frac{1}{n}\sum_i |B(a_i, r_i)|(1-\epsilon) = \frac{(1-\epsilon)}{n}\sum_i |B(a_i, r_i)|.$$

Since no attack works on replicas from two different partitions,

$$\sum_i |B(a_i, r_i)| \leq \mathrm{k}\ ,$$

where ℓ is the total number of attacks in the salvo. Thus, the probability that a replica will be subverted by a single attack is less than

$$\frac{\mathrm{k}(1-\ \epsilon)}{n}.$$

Thus, (1) becomes $2^{1-n(1-(\ell/n)(1-\epsilon))/2}$, or just $2^{1-(n-\ell(1-\epsilon))/2}$. Solving for the number of attacks, as before, we find that the attacker needs at least

$$\frac{n-2+2\log(\delta)}{1-\epsilon}$$

attacks to have confidence δ of subverting all replicas. If this confidence is reasonably large, then the number of necessary attacks depends linearly on the number of partitions. (If the confidence is small, the bound becomes negative, and that means that our framework is unable to guarantee that the attacker will need even a single attack to reach that low level of confidence.)

(Note that increasing the number of such partitions may be much more difficult than increasing the number of replicas. This is true, for example, if each partition represents a different operating system.)

3.6 The Vapnik-Chervonenkis dimension of an attack set

At this point, we would like to make some generalizations about the quantity $|B(s, r)|$ (recall that this is the number of different ways the attack salvos in the set s can behave when applied to the replicas in the set r). This number is, in some sense, a measure of how diverse the attack salvos are with respect to these replicas, but so far we have only given specific examples where $|B(s, r)|$ could be determined from the assumptions of the example.

We would like to say something more general about this value, or, more specifically, about its expected value, since this is what appears in our calculations when we bound the probability of all replicas being subverted (inequality (1)). Unfortunately, it is somewhat difficult to draw conclusions about $\mathbf{E}(|B(s, r)|)$ outside of a specific context, since it depends on the probability distribution used when randomly selecting replicas. However, (1) is an inequality, and the sense of the inequality is preserved if we upper-bound $\mathbf{E}(|B(s, r)|)$ by

$$\max_{r \in R^n} |B(s, r)|, \tag{4}$$

where R^n denotes the power set of all sets of n replicas.

Because of the way behavior strings are defined, $|B(s, r)|$ can be as large as 2^n, where n is the number of replicas in the set r. In other words, if there are sufficiently many attack salvos in s, and if they are sufficiently diverse. Moreover, adding a new replica might simply increase $|B(s, r)|$ to 2^{n+1}, and, in terms of what we can prove using (1), the defender gains nothing in this case. (In physical terms, this means that the attacks are so numerous and diverse to begin with that any new replicas are simply subverted along with the old ones).

However, if the number of behavior strings is less than 2^n, adding new replicas may be fruitful. Indeed, it is easy to see that if $|B(s, r)|$ is less than $2^{|r|}$, then any new superset r' of r will also have $|B(s, r')| < 2^{|r'|}$; this is because adding a new replica can no more than double the number of behavior strings. In fact, it can be shown (C.f., [26]), that if (4) is less than $2^{|r|}$, then $\max_{r' \in R^{n+i}} |B(s, r')|$ is less than

$$e^{(r+i)H(r/(r+i))}$$

for all $i \geq r$, where H is the binary entropy function; $H(x) =^{\text{def}} -x\log(x) - (1-x)\log(1-x)$. This function grows rapidly as i increases, but $2^{-(r+i)\eta/2}$ decreases more rapidly, leading to a decrease in (1). It follows that adding new replicas is ineffective up to a certain point only, and this is the point where $\max_{r \in R^n} |B(s,r)|$ first drops below $2^{|r|}$. After that, adding new replicas may decrease the attacker's chances of subverting the entire system, and after doubling this number, the attacker's chances begin to decrease exponentially.

The crucial threshold, where the maximum number of behavior strings is no longer 2 raised to the number of replicas, is the Vapnik-Chervonkis dimension of the attack salvo, and we will denote it with V_s. We have taken this concept from the theory of statistical machine learning, and using results from that area we can show that the probability of subverting all replicas is less than

$$\left(\frac{en}{V_s}\right)^{V_s} 2^{-n\eta/2} \tag{5}$$

if $n > 2V_s$.

3.7 Conclusion

In this paper, we have developed a framework for examining the benefits of redundancy in software systems that are under malicious attack. Such systems must generally be diverse, in the sense that the redundant copies do not succumb to the same attacks. Therefore there is some expense associated with redundancy; one cannot achieve the necessary diversity by simply making copies of an existing program. The central question is whether or not redundancy slows down malicious attacks enough to make it worthwhile. The goal of the framework developed in this paper is to show how various factors determine the return on investment achieved through redundancy.

To avoid having to make *a priori* assumptions about the attacker's strategy, our framework deals with *randomized* diversification strategies, which is to say that the process for creating redundant replicas has random elements. This leads to a natural measure of diversity with respect to a given attack or set of attacks: we measure diversity as the probability of creating a replica that is immune to the attacks in question. This is appealing because such a probability is also a natural way to describe the quality of the diversification process itself, so an immediate, quantitative link is formed between the quality of the diversification process and its effectiveness in retarding malicious attacks.

We have attempted to create a framework that is as flexible as possible, and it was used in a number of examples to examine how various factors determine the effectiveness of a diversification scheme. Although there are situations where diversification does not appear to work well, we have outlined some circumstances where it does work; these include at least some cases where there is an effective intrusion response system (which may consist of patching vulnerabilities between attacks), the case where the defender never creates two replicas that are vulnerable to any of the same attacks (hyperdiversity), and the case where the attacker only replays existing attacks.

Although we are able to say, to some extent, what characteristics should be sought in a redundant system, we have not said anything about how those characteristics might be obtained. In particular, it is not clear how to create redundant replicas that have high diversity, nor in fact whether it is even possible to do so without knowing what the attacker will do. Therefore, it does not seem that our framework can be used to predict how long a particular redundant system will withstand a particular series of malicious attacks. It can, however, be used to say what factors affect the performance of a redundant system, and how the performance might be improved when other circumstances remain the same.

4 Details of the Mutation System and Experimental Results

Aside from the analytic investigation discussed in Section 3, our project also involved an empirical investigation of diversity. In particular, we proposed an application of *software mutation*, a software engineering paradigm where programs are altered to simulate faults. Normally, this technique is used to determine whether software tests can uncover the simulated faults, but in this project, our purpose was to automatically create diverse replicas of software systems.

However, software mutation is a much broader paradigm than the simple creation of faults or diversity. Indeed, any software modification might be called "mutation," although we reserve the term for modifications that can be applied to many programs, not just one. Following the conventions of the software mutation literature (see [1, 5, 10] for example), we will say that a mutation system defines a set of *mutation operators* which map the set of possible programs to itself. Mutation operators could, in principle, facilitate many tasks involving software modification:

- Generalized software patches: In computer security, there is currently a need to retrofit programs with simple, well understood mechanisms that enforce simple security paradigms. A simple example is protection against buffer overflows. Although many different programs could be patched using just one basic principle, such as "place sentinel data on the stack to check for buffer overwrites," "avoid copying data without checking the length of the data," or "avoid using stack-allocated variables," existing software patch programs are text-based and thus tied to specific programs. However, if such general principles were regarded as mutation operators, and if they could, in fact, be performed by an automatic mutation system, the process of patching flawed programs could be greatly simplified.

- Software modification for execution monitoring: In many applications it is useful to monitor the execution of a program. Other types of execution monitoring have been used in the past to determine whether a process has been corrupted and also for intrusion detection [11, 12]. The insertion of data watchpoints into the source code of a program can also be viewed as a mutation operator.

- Software modification for fault injection: Fault injection is the deliberate introduction of errors into a computation at runtime. It has been used extensively for software analysis, including checks of safety and security properties [27, 28]. These techniques generally require instrumentation of a program's source code at the points where errors are injected. Although this instrumentation is usually done with a specialized system, it could also be performed by mutation operators if it were considered too expensive to build or buy such a specialized system.

Of course, regarding such software modifications as mutation operators has little significance unless we have a mutation system that actually implements them. To our knowledge, no existing mutation system is so general, and one of our goals was therefore to construct a general-purpose mutation system capable of implementing arbitrary mutation operators. In this section, we give the details of our system and report on our experiences when using it.

4.1 Mutation using Mutation Scripts

Our mutation system is driven by scripts that describe how each program is to be mutated. Conceptually, a mutation script must specify two kinds of things: *what parts* of the mutated program are to be modified, and *how* they should be modified.

Our mutation scripting language accomplishes this by using *mutation templates*, which specify what portion of a parse-tree is to be mutated by referring to the productions in the program's grammar. Each template is associated with code that modifies the parse tree, referring to the productions mentioned in the template. It is also convenient to have a body of code that is executed as soon as the mutation script is executed, in order to define functions and initialize global variables.

The details of our mutation scripting language will be given in what follows. First we will describe the language itself and its interface with the parse tree, and then we will describe the mutation templates we use to specify where mutations will take place.

4.1.1 The mutation language

Our mutation language is a functional language, similar to LISP. This decision was made for two reasons: First, the interpreter is easy to implement, and second, the dynamic scoping of functional languages seems well suited for our method of mutation using mutation templates and mutation scripts, since it simplifies the passing of parameters to parts of the script that are bound to mutation templates (this will be illustrated by an example later).

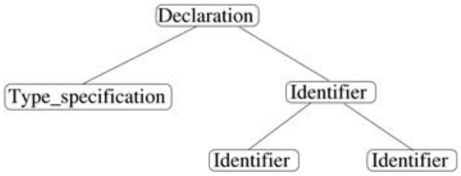

Figure 1: The template `#< ($type) ($identifier_lisr) >#` matches two different combinations of nodes. In practice, we find that the author of the mutation script probably only had one of them in mind.

The Mutation System script language, mscript, is used to specify the actions undertaken by the system. The language provides standard programming capabilities via a functional programming language similar in syntax to LISP, after which it was originally modeled. In addition to typical programming capabilities, mscript allows a user to specify a template which represents a tree data structure. The Mutation System uses the templates to find instances of the template sub-tree in the Input Tree provided to the system. If a template "match" is found, a rule specified in mscript syntax is executed. This rule may potentially modify the tree, changing its structure, or simply perform some type of analysis. A typical use of the language is to specify modifications to be made to a parse tree representing source code of a high-level language such as Java or C++. For example, templates which potentially match portions of the AST are defined and are "associated" with rules which generate a subtree to replace the "matched" subtrees.

A template takes the form of a regular expression whose alphabet symbols represent the terminals and nonterminals in the parse tree. Thus, for example, the template
`#< typename ($identifier ",")* $identifier "," ($identifier ",")* $identifier ";">#`
matches all but the the last identifier in an idealized C/C++ declaration that consists of a typename, followed by a comma-separated list of identifiers and a semicolon. The final identifier would need to be matched by a separate template.

Each template is associated with code describing how the source code matching the template is to be mutated. Since this code may reference variables in a higher scope, it is necessary to define the order in which matches are processed. In the current system, this is accomplished by a process that, intuitively speaking, only processes the first match found during a preorder traversal of the parse tree. (The actual procedure is somewhat more complicated and will be described below.) The program can use one of several mechanisms to prevent the same parse-subtree from being matched twice; most commonly, the nodes of the matched subtree are simply renamed during mutation. Thus, all matches in a parse tree can be found by iteratively calling the matching algorithm until no more matches are found.

The above explanation is "intuitive" because it is ambiguous. For example, in the parse tree shown in Figure 1, the template `#< ($type) ($identifier_list) >#` could match at two different node-combinations, going only by the description given above. (This can happen even when the underlying grammar is not ambiguous itself.)

The problem is that regular expression syntax, which we adopted because it seemed like it would be readily understandable to users of the mutation system, is not meant to be applied to trees, and so we are abusing that syntax here. We resolved this problem by establishing three conventions, which are meant to make the system do what the user expects. When using the system, we found that it did, indeed, behave in an intuitive way, which is to say that we did not find ourselves explicitly thinking about the additional matching conventions when we wrote mutation templates.

We now describe our additional matching conventions:

1. In regular expressions, the '`*`' operator implies a sequence of symbols, so we do not want it to match a node while also matching the children of that node. In other words, the template above should not match node sequence $\langle 2, 4, 5\rangle$ since this is counterintuitive and would lead to confusing programs.

2. Matches occur according to a postorder left-to-right traversal, which is to say that the first `identifier` matched by the template `#< ($identifier)* >#` will be the first one encountered in a left-to-right

postorder traversal, the second `identifier` will be the second one encountered in the same traversal (excluding ancestors of the first match, as required by the first convention) and so on.

3. These conventions eliminate ambiguities caused by matching a parse tree instead of a string, but some ambiguity still remains because our regular expressions can correspond to nondeterministic finite automata. This ambiguity is resolved by the convention that the longest possible string of symbols is always matched, subject to the first two conventions.

Mutation templates are associated with code that mutates the portions of the parse tree that are matched. To facilitate this, an alphabet symbol S can be replaced by the construct `(<varname> = $`S`)` (recall that the alphabet symbols are the names of terminals or nonterminals in the grammar). Here, `<varname>` represents a variable name that will be bound to whatever parse-tree node matches S, and then made available to the mutation code.

A final feature of templates is that they allow explicit matching of parse subtrees having a certain structure. An alphabet symbol S in a template expression can be replaced by the construct S`(<template>)`, which only matches a subtree if its root is of type S and the rest of the tree can be matched to the mutation template `<template>`. This construct overrides the convention that one symbol in a mutation template not match a node in the parse tree at the same time that another symbol matches an ancestor of that node. For example, the template

```
@PrimaryExpressionNode((.)* (id = $IdentifierNode) (.)*)
```

matches any node labeled as a `PrimaryExpressionNode`, provided that it is the root of a subtree containing at least one `IdentifierNode`.

4.1.2 The lexical structure of *mscript*

The lexical structure of mscript is based on the ASCII character set. An arbitrary amount of white space separates lexical elements. A string is a combination of alphanumeric characters. A constant is an integer, or a floating point number, or a sequence of characters other than an end of line enclosed in quotes, for example ”some string”.

The following characters have special meaning in the mscript language:

- A ’`;`’ denotes the beginning of a comment line. Comments may begin anywhere in a line. A comment is completed by an end-of-line or end-of-file character.

- Parentheses, ’`(`’ and ’`)`,’ delimit an element of type *list* in mscript.

- The empty list, ’`()`’ denotes the element *nil* which can also be represented as ’`nil`’ in mscript.

- The symbols ‘`#<`’ and ’`>#`’ enclose an element of type template (used to specificy which parts of a parse tree are to be operated on).

- Function calls are represented as ’`name.arg`,’ where ’`name`’ is the name of the function, and ’`arg`’ is the argument, which may be a list. Thus, ’`.`’ is a special character.

- Brackets, [and], delimit a lambda expression.

- An apostrophe, ’`'`’, immediately preceding an object suppresses evaluation of the object, effectively making the object a constant. Thus, ’`'var_name`’ refers to the variable itself, rather than referencing the value of the variable.

4.1.3 The syntactic structure of the mscript language

Our mutation system is driven by scripts that describe how each program is to be mutated. Conceptually, a mutation script must specify two kinds of things: *what parts* of the mutated program are to be modified, and *how* they should be modified.

Thus, our mutation scripts have two main types of components: *mutation templates* that specify what syntactic constructs will be modified by a given mutation operator, and *mutation code* that does the actual modification, referring to the syntactic elements identified by the template. For example, a simple mutation operator might be specified as follows:

```
#<type_specifier identifier* ($id = identifier) >#
```

(mutation code)

The template matches any type specifier followed by one or more identifiers, and associates the variable `id` with the rightmost identifier. The rightmost identifier can be accessed by the mutation code via the variable `id`.

The mutation system works by modifying parse trees, and hence the terms in the mutation templates, such as "identifier" and "type_specifier" in the example above, refer to nonterminals in the parse tree.

The matching templates are based on regular expression syntax, which makes the templates easier to write and understand than, say, a context-free grammar. However, regular expression syntax is not suited for matching portions of a tree, so we use an augmentation wherein a node may be specified by its structure, as well as by its identifier. Specifically, an identifier in a template can be replaced by a nested regular expression, which must be satisfied by the parse subtree rooted at the node in question. Obviously, this nested regular expression can contain more nested regular expressions. Nested regular expressions are enclosed by "`@..(...)`" in the following example:

```
#< (assg = @AssignmentNode(
            (lhs = @PrimaryExpressionNode(id = $IdentifierNode))
            {"="}
            [$BinaryExpressionNode
             $PrimaryExpressionNode
             $NonLogicalExpressionNode
            ])
    )
>#
```

This template matches any parse tree node whose identifier is `AssignmentNode`, as long as the subtree rooted at this node matches the regular expression

```
#<PrimaryExpressionNode
  {"="}
  [$BinaryExpressionNode
   $PrimaryExpressionNode
   $NonLogicalExpressionNode]
>#,
```

and as long as the tree rooted at the `PrimaryExpressionNode` in this expression matches the regular expression `#< $IdentifierNode >#`. Here, the square brackets imply disjunction. Note that the `AssignmentNode` at the top level is associated with the variable `assg`, the `PrimaryExpressionNode` is associated with the variable `lhs`, and the `IdentifierNode` is associated with `id`.

The mutation code associated with such templates is written in a simple `LISP`-like language. This allows an initially simple interpreter to be extended as necessary. We found, not surprisingly, that programmers had difficulties with a pared-down functional language such as this. At the same time, we did not discover any specialized requirements for this language that justify a custom interpreter, although support for nested function definitions turned out to be useful in practice. Still, the mutation language in our prototype was not a complete success, so in this paper we will not describe it in detail. We simply note that the mutation language accesses the parse tree via variables defined in the mutation template, and has access to several basic functions for modifying the parse tree. These functions create new parse-subtrees, delete parse-subtrees from the parse-tree, and insert new subtrees. These functions also allow changes to the identifiers associated with parse-subtrees, and, among other things, this mechanism can be used to stop a template from matching the subtree twice. Thus, nearly any programming language can be used as the basis for generalized mutation operators.

The syntax of the mutation language, which we named mscript, is essentially the same as the syntax of LISP, with several exceptions.

Function call syntax: In LISP, a function call has the form `(function-name     arg1 arg2 ...  argn)`. The corresponding call in mscript is

```
function_name.(arg1 arg2 ... argn)
```

Having this different function call syntax leads to several other differences as well. For example, it is possible to say

```
Set.('arglist '(arg1 arg2 ... argn))
Function.arglist
```

and get the same effect as calling the `Function` on the original argument list. It is also possible to use `a` single variable as an argument to a function, as in

```
Function.var
```

even if that variable is not a list.

Variable assignment: In mscript, the function `Set` defines a variable in the current scope, and gives it a value. Currently, things don't work well if there are two or more `Set`s of the same variable in the same scope. This is a bug, but `Set` should be thought of as a declaration with initialization, and not an assignment. For example

```
Set.('x 12)
```

declares a variable named `x` and initializes it with the value `12`.

Assignment is done with the function `Reset`, which searches for the innermost declaration of a new variable, and gives it a new value. For example,

```
Reset.('x 34)
```

assigns the value `34` to the variable `x`, assuming that `x` was declared in some enclosing scope with a `Set`.

Built in functions: Many built-in function that are so common that they're essentially a part of LISP aren't currently part of mscript. In particular there is not currently a `defun` function. Instead, functions are defined by assigning a variable to a lambda expression, as in

```
Set.('fcn '<lambda_expr> )
```

which defines a function named `fcn` whose body is the lambda expression.

Lambda expressions The lambda function is the function that is used to do function calls. Lambda's arguments are a function and a parameter list, and its purpose is to call the function using the parameter list. For example, in lisp one might say

```
(lambda Add (2 3)),
```

which would cause the function `Add` to be executed with the arguments `2` and `3`. In mscript, the corresponding call would be

```
Lambda.(Add (1 2)).
```

In both mscript and lisp, the function is a list whose first element is the parameter list, and whose second argument is the function body. For example, the list

```
((a) Add.(a 1))    <mscript>
((a) (add a 1))    <lisp>
```

can be interpreted as a function (through the use of lambda) that takes a single parameter, `a`, and adds one to it. Therefore,

```
Lambda.('((a) Add.(a 1)) (2))       <mscript>
((lambda'((a) (add a 1))) 2)       <lisp>
```

has the effect of adding 1 to 2 and returning 3. First, the `2` at the end of the list is bound to the variable `a`, and then the function call `Add.(a 1)` is executed.

When a list is evaluated in LISP, it is automatically assumed that the first element is a function. If a LISP interpreter is asked to evaluate `(add 2 3)` it assumes that `add` is a function. The interpreter calls lambda automatically to evaluate the function (at least this is how it works conceptually). Therefore, since the list ((a) (add a 1)) can be interpreted as a function, its should be possible to say

```
( '((a)  (add a 1)) 2)
```

in LISP; this should execute the function defined by ((a) (add a 1)) in the parameter 2. (However, many LISP interpreters do not implement lambda so literally.)

In mscript, one can say

```
'((a) Add.(a 1)).2,
```

which has the effect of executing the function defined by ((a) Add.(a 1)) with 2 as the parameter. With LISP, the fact that a list is being evaluated implies in itself that the first element of the list is a function, and it triggers a call to lambda. In mscript, the appearance of a period between to elements implies that the item on the left side of the period is a function call, and it triggers a call to lambda as well. Therefore, it is not necessary to call lambda explicitly in either language (but lambda does exist in mscript).

In LISP, the function defunis used to define a function.

Parameter lists

Ordinary parameter lists: Mscript is technically based on a tree structure, not on a list structure like LISP is. Normally this makes no difference, because a mscript tree is more or less the same as a list of lists as far as the user is concerned. But it affects the way argument lists are parsed. In mscript, a parameter list can contain lists itself. In contrast, a lisp parameter list can only contain variables. For example, the following are legal lambda expressions in mscript:

```
(a Sum.(a 1))
((a) Sum.(a 1))
((a (b)) Sum.(a b))
```

Assume that these one of these functions is called Add (as it would be if we said Set.('Add '(a Sum.(a 1))), for example). The first expression should return 2 when called as Add.1, but if it were called as Add.(2), it would cause an error by trying add one to a list. If Add were defined to be the second expression above (e.g., Set.('Add '((a) Sum.(a 1))),) the the call Add.1 would result in an error, because Add expects its argument to be a list and not a number. Likewise, if Add were defined with Set.('Add '((a (b)) Sum.(a b))), then Add.(1 2) would result in an error, because Add would expect a list consisting of a number followed by a list. (The correct call would be Add.(1 (2)) in that case.)

Templates as parameter lists To (somewhat) extend the notion that parameter lists are trees that have to match the argument list, a mutation template can also be used as a parameter list. Such a parameter list has the form

```
#< template >#
```

where the template has the same form as other mutation templates. If a template appears in the parameter list of a function, then the corresponding argument is expected to be a parse subtree. The function body is executed once for *each* time the template is matched within the parse tree. The result is a list containing the result of each function evaluation. For example, consider the function tname defined as follows:

```
Set.('tname
        '(#< [tree] $1typedefname ># ($1typedefname)))
```

The function would be called with a single argument, which would be a parse subtree. The function body, which is ($1typedefname), is evaluated once each time a typedefname is found in the argument. The function body does not do any computation, but it returns the parse subtree that was matched as being a typedefname. Since all the values obtained from evaluating the function body are placed in a list and returned, the effect of tname is to return a list of all the typedefnames that are found in the subtree that is used as its argument.

(In the current language, one difference between a real mutation template and a function with a template parameter is that the real mutation template can be used to modify code, while a function with a template has no easy way to modify the code. On the other hand, a function with a template parameter synopsizes its results in a form that can be manipulated by other functions, and a regular mutation template cannot do this.)

Builtin Functions The currently implemented mutation functions are:

- `Eval(x)`: The argument `x` is a tree describing a mutant rule. `Eval(x)` evaluates this tree and returns the result. This function is described above.
- `Replace(n, x, y)`: The first argument is a number, the second and third are trees. The result is a tree formed by taking each child of the first tree and using it to replace the `n`th child of the each child of the second tree. (It is useful to think of the second and third arguments as lists rather than trees, the members of the list correspond to the children of the tree. Then the third argument is a list of lists. Each member of the list `x` is used as a replacement for the `n`th member of each list in `y`.)
- `Cond.((c1 a1) (c2 a2) ... (cn an))`: If `c1` evaluates to something other than `nil`, return the result of evaluating `a1`. Otherwise, if `c2` evaluates to something other than `nil`, return the result of evaluating `a2`, and so on. If none of `c1, c2, ..., cn` evaluate to anything besides `nil`, return `nil`.
- `Lambda.(func, args)`: Similar to lambda in LISP. It is not often called directly, but knowing how it works is useful for understanding function calls. `func` is a list of of the form `params action params action ... params action`. `args` is a list of arguments. For each `params-action` pair, and for each argument in the list `args`, `Lambda` opens a scope, binds the argument to the parameter `params`, as described below, and evaluates `action`, and closes the scope. The handling of return values is currently somewhat bizarre because we have not decided what makes the most sense as the return value of a lambda expression from the standpoint of language usability. The return value is defined as follows: If any evaluation results in an atom, then the return value is the value of the last evaluation that resulted in an atom. If no evaluation results in an atom (that is, every evaluation results in a list) then all the lists are catenated and the result is returned.

  If we were simulating the traditional LISP form of lambda, `func`would consist of a single `param-action` pair and `args` would contain a single list. `params` would be a list with the same number of elements as the list contained in `args`. The first member of the `param` list would be bound to the first member of the argument list, the second member of the `param` list would be bound to the second member of the argument list, and so on, and then the action would be evaluated.

  Our version of `Lambda` does this repeatedly, once for each `param/action` pair and each argument. However, if a parameter cannot be bound to an argument (for example, if they're lists with different numbers of elements), `Lambda` does not return an error; instead it simply skips the evaluation of that particular `action` for that particular argument.

This behavior, together with the method we use to bind parameters to arguments, makes the function call mechanism extremely powerful and fun to use, though somewhat tricky.

An argument and the corresponding parameter are both single items, as implied by the use of the singular case being used to describe them. The parameter can be an atom, a list, a tree, etc, but the structure of the argument must match the structure of the parameter enough to allow a variable binding.

If the parameter is `mylist` and the argument is `(hello world)`, then `mylist` will be bound to the list `(hello world)`. If the parameter is the list `(x y)` an the argument is the list `(hello world)`, then `x` is bound to `hello` and `y` is bound to `world`. If the parameter is the list `(x (y))`, and the argument is the list `(hello world)`, then the match fails because we expected the second element of the argument list to be a list itself, and instead it was the atom `world`.

The association of parameters to arguments is accomplished by simply placing the string that is given as a parameter into a symbol table, and making the argument into its value. This is done in a local scope, so the association disappears after the function call.

To make things interesting, a special feature exists that allows a parameter to be a mutation template. In this case the expected argument is a parse subtree. If the template matches the subtree, then replacement references are resolved (see above) before the action is evaluated, so the replacement references act like symbols in the symbol table at the local scope. Their values are the corresponding parse subtrees. If the parse tree matches the template in more than one place, the action is evaluated once for each match.

An example:

```
Set.('collect_conditions
'((\#< [tree] when_clause
     when $1cond => $1select_wait_alt >\#) ($1cond)))
```

defines a list that can be used as part of the `func` list in a `Lamdba` expression. The section `#< [tree] @when ... >#` is a template (see above) that is matched when we traverse a parse tree and we hit a node of type `when_clause` whose children are a node of type `cond` (in this case a specific condition in the code being analyzed). When a match occurs, the variable `$1cond` is bound to the parse subtree that represents the condition in question.

For each match that occurs, `Lambda` evaluates `($1cond)` and places the results on a list. The result is a list of all the conditions that appear in `when` clauses of the parse subtree that was passed as as an argument.

- `Head.x`: Head is analogous to `car` in LISP. If `x` is the list `(hello world)`, then `Head.x` evaluates to `hello`. Likewise, `Head.(hello world)` evaluates to `hello`. But `Head.(x)` evaluates to `(hello world)`, because it returns the first member of the list `(x)`, which is `x`, and evaluates to `(hello world)`.
- `Tail.x`: Tail is analogous to `cdr` in LISP. It returns the list that results from removing the first element of `x`. If `x` is the list `hello world` then `Tail.x` evaluates to `(world)`. If `x` is `(hello)` then `Tail.x` evaluates to nil. If `x` is `goodbye cruel world` the `Tail.x` evaluates to `(cruel world)`.
- `Or.(c1, c2, ..., cn)` Return `t` if any of `c1, c2, ..., cn` are non-nil. `Or` does short-circuit evaluation.
- `And.(c1, c2, ..., cn)` Return `t` if all of `c1, c2, ..., cn` are non-nil. `And` does short-circuit evaluation.
- `Not.(c)`: Return `t` if `c` evaluates to nil; return nil if `c` does not evaluate to nil.
- `Minus.x`: makes `x` negative. Anything can be negative but at the moment this does not mean anything semantically unless the negative thing is a number.
- `Sum.(x1 x2 ... xn)`: The arguments `x1, x2, ..., xn` are assumed to be numbers. `Sum` adds them. Subtraction is accomplished with `Minus`, that is, `Sum.(x Minus.y)` subtracts `y` from `x`. `Sum.(Minus.y x)` has the same affect (because addition is commutative. Note that `Minus` is not an infix operator).
- `Num_Equal.(x1 x2 ... xn)`: test for numeric equality. Returns `t` if `x1, x2, ..., xn` are all the same number, otherwise returns nil.
- `Set.(a b)`: Associates the name `a` with `b` in the symbol table at the current scope. For example, saying `Set.('hello 'world)` means that `hello` will evaluate to `'world` until the scope is closed. Note that the antecedent of a mutant rule has its own scope. Therefore, a `Set` in the antecedent of a mutant rule will not have any affect after that antecedent is finished being evaluated. `Set` is not so much an assignment statement as it is a declaration with an initialization.
- `Reset.(a b)`: Searches the scope stack for the first occurrence of `a`. The old value of `a` is removed from the symbol table, and `b` becomes the new value of `a` *at the level of scoping where* `a` *was found.* Therefore, using `Reset` is like making an assignment to a variable that has been declared using `set`. (As expected, `Reset.(a b)` results in an error if `a` is not in the symbol table, that is, if it has not been declared by a `Set` in an ancestor scope.)
- `OpenScope`: Open a symbol table scope. This allows static scopes to be created without kludging.
- `CloseScope`: Close a symbol table scope. This allows static scopes to be created without kludging.
- `Defined.(x)`: Look up the value of a symbol. If the symbol exists, return its value. If it does not exist, return nil.
- `Reverse.x`: Reverses the elements of the list `x`.

- `String.x`: Returns a string consisting of the terminals encountered during a preorder traversal of the argument.

- `Consume.x`: Always returns nil. The reason for having this is that currently, mutation is done by evaluating the right side of the mutation rule and printing the list that results. If we have to do some computation that should not appear in a mutant, we put that computation inside a `Consume`.

- `Message .(x)`: Print out whatever is in the variable `x`. Unfortunately, Message is a bit fragile because of the strong assumptions it makes about the format of the argument. For example, `Message.((x))` currently causes a core dump regardless of what `x` is, because `Message` does not expect a list containing a list. The recommended way to call Message is

```
Message.(String.<arg>)
```

- `Emit.x`: Prints its argument to the file containing the instrumented code. `Emit` is more robust than `Message` because it accepts tree-structured arguments and also knows about variables of the parse-tree type (because of some implementation weirdness, parse subtrees are their own type of object even though objects in the mutation language are trees too.) The problem with `Emit` is that it will not work until the output file has been opened, meaning that it only works after mutation has started. In other words, `Emit` can only appear in a mutation template.

- `PrintArg.x`: Prints its argument to stdout. `PrintArg` is more robust than `Message`, but it is meant mostly as a debugging function. When it encounters a parse subtree, it prints the contents inside square brackets.

4.2 Practical Drawbacks of Metasoftware Engineering

In our system, generalized mutation operators must be programmed; the operators are described by mutation scripts instead of being hardwired into the mutation program. If it were otherwise, the result would not be a general-purpose system. The programming of mutation scripts therefore becomes a software engineering task, and we have found that the scripts can easily be complex enough to justify this analogy. Unfortunately, the development of mutation operators involves a number of issues not encountered in traditional software engineering tasks, and we will discuss those problems here.

As an example, consider the problem of modifying a program so that all local variables are allocated from memory, instead of being allocated from the stack. Such a modification would prevent most buffer-overflow attacks from succeeding.

The most obvious element of such a mutation operator is to change the way local variables are declared; for example, the `C` declaration `int c` should become `int *c = (int *)malloc(sizeof(int))`. For such an operator to be viable, however, it must also free the locally allocated variables when they go out of scope, since otherwise the mutated programs might not be usable replacements for the original ones.

Programmers often believe, initially, that such mutations can be made with programs like `sed` that scan and replace text but do not parse the program. This leads to an explosion of special cases, as when it becomes necessary to ensure that `int c` is not replaced with `int *c = (int *)malloc(sizeof(int))` when the statement is part of an aggregate type declaration instead of being a local variable declaration.

In fact, similar problems occur in subtler ways even for simple mutation operators, such as the replacement of one arithmetic operator with another in the mutated program. In `C`, for example, the set of legal replacements for an operator depends on the data-type being operated on, since pointers may not subjected to multiplication or division. Thus, the mutation system must not only find operators to replace, but know the data-types of variables, which leads to immediate problems for text-based systems.

It was such problems that motivated our use of mutation technology based on parse trees. We began the current project with a rudimentary parse-tree based system capable of performing the simple mutations generally associated with software mutation analysis, such as operator replacement and variable replacement. For such mutations, the parse-tree based approach seems to successfully overcome the problems associated with text-based replacement.

Unfortunately, the parse-tree based system also runs into problems when there is a need to program more complex mutation operators. Like text-based approaches, it is based on the syntax of a program rather than the semantics. Or, to be more precise, it provides some semantic information — more than a text-based approach — but that information has limited utility in the programming of complex mutation operators.

Consider again the example above, where the goal was to prevent local variables from being allocated on the stack. The first problem faced by the mutation programmer is identifying local variable declarations in the parse tree, so that they can be modified. This task is easier than it would be with a text-based system, because the grammar contains a well-defined set of productions for parsing variable declarations, and those productions are reflected in the parse tree. However, the programmer must discover *which* parse-tree patterns can denote a local variable declaration in order to use the matching techniques described in Section 4.1.1 (or any other parse-tree based modification technique). To make matters worse, there are many special cases, because different declarations need to be modified in different ways. For example, `d` in the declaration `int c, d;` needs a different modification than the `c`, because the data-type, which is needed to perform the mutation, is elsewhere in the parse tree with respect to the variable.

This particular problem, and many others, can be overcome by clever design decisions, but those decisions are intimately tied to the structure of the parse tree. In contrast, most software development projects that have abstract, high level designs do not depend on such excruciatingly small details as those found in a parse tree.

At this point, we can only say for certain that the development of mutation operators is different from standard software engineering, at least with our system. Since no one is experienced in this particular type of software engineering, we can expect the process to have rough spots that would not exist in a more traditional setting. However, we suspect that syntactic approaches — both text based and parse-tree based systems — may inherently leave something to be desired. It is often natural to describe mutation operators by saying how the *semantics* of the program should be changed, but that forces the mutation programmer to find *all* combinations of syntactic constructs that could lead to the behavior in question. In contrast, a software developer on a standard project only has to find *one* combination of syntactic constructs that leads to the behavior in the specification. The resulting problems may make syntax-based mutation scripting inherently impractical for mutation operators meant to bring about particular changes in a program's behavior.

5 Creating diverse systems with abstract interpretation

The goal of diversification, as we have stated it, is to have a number of components or servers performing the same function, so that the defender is not deprived of that function if some components are maliciously attacked. The backup components have to perform the same function as the original, so they should have the same specification, but components of the type in question often lack formal specifications, so we must make the replicas based on the source code of the original program, doing our best to to preserve the important aspects of the original semantics.

The desired situation is described in Figure 2. S is the set of possible program semantics, and the mutation operator M maps this set back to itself. The semantics of the mutant are not strictly the same, but they preserve the behavioral features that are needed to perform the intended functions of the component.

If we compare this diagram to what is actually implemented by the system described in Section 4, we can refine it further. This is shown in Figure 3. Each mutation operator picks out some aspect of the semantics using mutation templates (the mapping M_1 in the figure), transforms it (the mapping M_2), and finally installs it in an operational program (the mapping M_3). Mappings like M_1, and also M_2 if it preserves the original semantics, are one of the main concerns of *abstract interpretation* (see [9]), a field closely related to symbolic execution (C.f., [6,8]) and partial evaluation (C.f., [14]). In general, abstract interpretation simply consists of abstracting some aspect of software semantics an interpreting it.

How the abstraction is done, and how the resulting semantic information is used, depends on the application. For example, a system proposed by [20] tries to find upper and lower bounds on the ranges that variables might have when a program executes; this system and others like it are often called *symbolic execution* systems but symbolic execution is, in turn, a form of abstract interpretation. Symbolic execution has also been used to find algebraic representations of program semantics, again with the object of test data generation [7, 22]. Here, the idea is that if the program can be reduced to a symbolic form that can

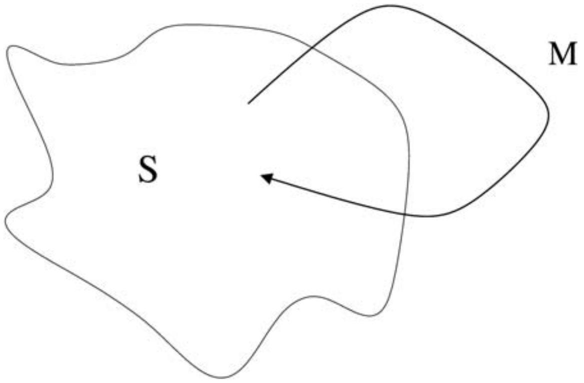

Figure 2: Illustration of a mutant operator as a mapping from the set of possible program behaviors to itself.

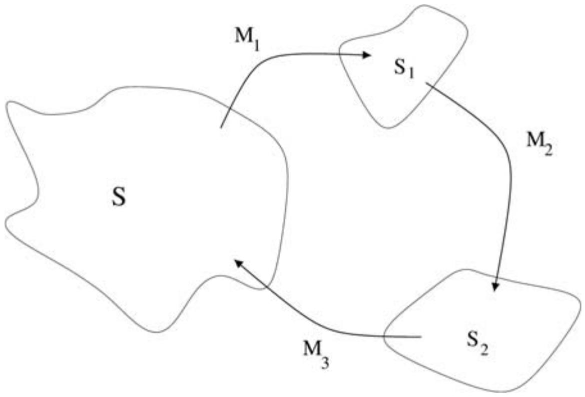

Figure 3: The mutation process resolved into three stages: the mapping of (part of) the source code to an abstract representation (M_1), the modification of the semantics thus represented (M_2), and the reinstallation of that semantic behavior into an executable program (M_3).

be algebraically manipulated, one can find tests that satisfy certain predicates; one simply says what those predicates are, plugs them into the symbolic representation of a program, and solves for the input variables. In cite [18], a similar paradigm is discussed, but there, faults are introduced into the abstract representation of the program's semantics, and one tries to find input values that trigger these faults. Once again, this is to be accomplished by essentially solving for the input variables. In addition, symbolic execution is used in many mutation systems to try to ensure that the mutants, which are supposed to have new semantics, do not in truth have the same semantics as the original program; C.f., [3]; here, the abstract semantics of the mutated program are just compared to those of the original program. (The description of our own system, given below, will serve as a more concrete example of the ideas involved.)

The emphasis in abstract interpretation is preserving some aspect of a program's semantics, and in this sense it contrasts with mutation, where the emphasis is on changing behavior. This, and the fact that abstract interpretation is a richer science than software mutation, suggests that abstract interpretation might form a better basis for software diversification.

In principle, abstract interpretation has little to say about *how* semantics are manipulated; this issue is left to specific abstract interpretation techniques. In practice, however, systems that perform abstract interpretation have a very different flavor from mutation systems. The emphasis is on abstract representations of program semantics that support formal reasoning and algebraic manipulation.

Our goal in this section is not to mimic mutation using an abstract interpretation system, but rather to step back and examine the application of abstract interpretation to the problem of software diversification. Referring to Figure 3, we can cite a number of advantages with such an approach.

1. If the semantic representations in S_1 or S_2 can be interpreted, then they, in themselves, can form the core of a diverse system. The interpreter itself copies the behavior of the original program when it is executed on the abstract representation. If the program deviates from its intended behavior, but the interpreter is unaffected, we can not only detect but correct the deviation in the original code.

2. The mapping M_2, which is where the actual mutation takes place, is based on semantics, not syntax. Based on the experience reported in Section 4.2, this can be a significant improvement on, or at least a useful augmentation to, syntactic mutation techniques.

3. We can also use M_2 to choose what aspects of the semantics we concentrate on, because the semantic representation representations generally used in abstract interpretation support such abstractions easily. For example, if we are interested in buffer overflow attacks, we might abstract the semantics pertaining to the contents or location of character buffers. We can use M_1 for this as well, and, in fact,

the system described in Section 4 does exactly that when it searched for subtrees matching a certain template. But once again, the approach of Section 4 is syntax-based and thus leaves something to be desired.

4. As a side effect, the ability to get abstract representations of software behavior might help us when we search for vulnerabilities or analyze attacks.

Abstract interpretation techniques generally have two important drawbacks. First, the abstract representations can be voluminous and thus tax system resources. Second, some semantic information is hard to translate into the desired abstract form (a notable example being behavior that results from pointer aliasing). A more practical issue is that the programs we are interested in here are procedural or object-oriented, which is a representation quite different from the one we seek. This means that the implementation of a general-purpose abstract interpretation is time-consuming due to the sheer number of diverse features that must be implemented.

All of these limitations can be sidestepped in a system that uses abstract interpretation to implement diversity. Instead of insisting that S_1 represent the semantics of whole programs, we can use a number of different mappings in place of M_1 to extract the semantics of small sections of the program. The size of the abstract behavioral descriptions can be bounded by limiting the size and number of program chunks whose semantics are represented. Program features that cannot be represented, or ones that require features not yet implemented in the abstract interpretation system, can simply be left out of the process by choosing program chunks that do not span such features (for example, if loops are not supported, we might treat the body of the loop, a section of code before the loop, and a section of code after the loop as three distinct code-chunks.)

At any given time, we can randomly select which parts of the program we represent abstractly. This is in accordance with the principle of randomization outlined in Section 3. An attack may go unattended because it does not exploit the semantics being mapped, but since the choice of what to map is made randomly, there is a certain probability that the attack *will* affect part of the semantics that are being replicated. If we take some corrective action, then this probability is just the survival probability discussed in section 3. In fact, even if we just raise an alarm when the actual and expected program behaviors diverge (as we do in the system we will shortly describe), we still have a diverse system, though it is now the intrusion detector that is diverse, and fooling a particular intrusion detector is analogous to subverting it.

5.1 Details of the System for Extracting and Monitoring Behavioral Constraints

In this section, we describe our framework for extracting behavioral information from programs. In our original proposal, it was meant to help us in finding mutants that preserved the intended semantics of the original program, but it became the backbone of what we believe is a more viable approach to diversity, based on monitoring a program's compliance with semantic constraints. In this section, we will describe our system for analyzing program behavior, and our use of that system to create diversity in critical systems.

The original aim of our mutation system was to create only *equivalent mutants*, that is, mutants whose semantics were (superficially) the same as those of the original program. The motivation for this is that, in reality, the semantics elements of a program can be divided into *intentional semantics* and *incidental semantics*. The intentional semantics describe the behavior that must be present to fulfil a program's specification, and the unintentional semantics describe incidental behavior, such as the writing of temporary files, the specific choices of data structures, and, in principle, even such details as what memory locations are allocated and how much the program heats up the CPU.

The motivation for our approach to survivability was that malicious attacks often take advantage of incidental semantics. Indeed, such attacks are especially troublesome because they exploit vulnerabilities not apparent in the specification. If an attack relied on nothing but the semantics given in a specification, then the vulnerability could (at least in principle) be uncovered by a formal methods. Therefore, we intended to build a mutation system that could preserve the intentional sematics of a program, while altering the incidental semantics. In the language of mutation analysis, the goal was to create *equivalent mutants*.

Existing software mutation systems already have mechanism for identifying equivalent mutants, though this is done to avoid them and not to generate them. Our intent was to reverse the standard approach.

To identify equivalent mutants, it is necessary to extract semantic information from the program being mutated. The semantic information extracted by our system takes the form of *semantic constriants*, which are essentially Horne clauses asserting that if some set of preconditions is satisfied, then some postcondition is satisfied. Some preconditions are always true, some depend on the data in a specific execution of the program, and some are deduced using other Horne clauses.

If a function or program is completely specified in terms of semantic constraints, then we can, at least in principle, determine the high-level semantic properties of the program by combining the constraints. For example, if we were interested in the relationship between the input variables and the output variables of a function, we could simplify that function's semantic constraints and obtain a (hopefully concise) set of Horne clauses describing the output variables as algebraic functions of the input functions. We might also try to obtain a concise description of the possible control paths through the function, though the resulting constraints would be different from those obtained in the previous case. Such simplification can be characterized as *abstract interpretation*, which is defined as the mapping of some aspect of a program's behavior into a more abstract, hopefully more tractable domain (see [9] for a concise overview).

Usually, abstract semantic descriptions can be manipulated algebraicly, as our semantic constraints can. Therefore, we can use them to isolate the semantic preconditions and postconditions of some portion of a program that is to be mutated. If a potential mutant has weaker preconditions and stronger postconditions, then it will not change the part of the program's behavior represented in the constraints.

Several things became apparent as we implemented our system for extracting semantic constraints:

1. Constructing the constraint extraction and abstract interpretation systems was a more time-consuming task than we had expected. As the project progressed, we began to see long-term value in the development of a system that was more expressive than what we had originally planned. This prevented us from completing our orignally proposed system for identifying equivalent mutants, but for various reasons (see Sections 3 and 4.2) the mutation approach no longer seemed promising. Within the given resource constraints, we pursued an approach to diversity that seemed more promising, and which will be described shortly in Section 6.1.

2. Though it would be possible, at least in principle, to *identify* equivalent mutants, we could not construct a system to *create* them within the constraints of the current contract. Our original plan had been to automatically extract libraries of potential mutants from existing programs, maintaining the pre- and postconditions for each mutant in the library as well. Under this plan, mutation would be carried out by first extracting the pre- and postconditions of a piece of code that was to be mutated, and then search for an equivalent mutant in the library. However, we found that semantic constraints tended to be too detailed for this to work; in most cases, we would simply fail to find an equivalent mutant.

3. In principle, a system based on semantic constraints could emulate a diverse system based on software mutation. If some portion of a program were meant to be mutated, and we possessed the preconditions for that program fragment, we could monitor the program during execution and determine (a) when the mutated fragment was to be executed, and (b) the starting values of the variables used in the program fragment. In principle, therefore, we would have enough information to execute a mutated program fragment, and, with the postconditions in hand, we could compare the resulting data-state to that of the original program. If desired, we could even inject the data-state created by the mutant into the original program. Thus, by using constraints to trigger the execution of mutated code *fragments*, we could emulate the effects of a diverse, redundant system without actually running different programs in parallel.

4. If the information contained in semantic constraints were used to *directly* calculate the results of executing the corresponding code, this calculation would, in itself, constitute an equivalent mutant.

These considerations led us to a new blueprint for a diversification system based on constraint monitoring. The details will be given below, but the overall idea is to monitor the program's compliance with the semantic constraints that have been extracted from it. For the reasons given above, this is the same as having one equivalent mutant for each constraint being monitored. There are two important differences between this system and the one we had originally proposed:

1. Diversity is not created by having many mutants for a one or more large portions of the code. Instead, it is created by using a single mutant for each one of many small fragments of the code. This seems to us to be a more effective way to increase the survival probability discussed in Section 3.

2. The resulting system is self-contained; no external harness is needed to make redundant mutants run in parallel and compare their results. Such a harness was not included in our original proposal, so the advantage of this approach is a reduced need for external technology. The disadvantage, as we will see, is a potentially lowered resistance to some denial-of-service attacks.

Before delving into the details of our approach, a final issue that must be discussed is the feasibility of extracting and using semantic constraints. There are two hurdles: first, the number of semantic constraints extracted from a single program can be quite large, and second, it is not possible to obtain a complete but analytically tractable description of a program's behavior.

To see the problem posed by having many program constraints, the reader may imagine two programs running in parallel. One program is the real one, and the other is a sentinal program that checks the behavior of the first. At every execution step, the sentinal program reads the data-state of the real program, executes the next step on that data, and compares the results to those of the original program. Disregarding the fact that the sentinal may share the vulnerabilities of the original program, this setup is roughly equivalent to what our system would do if every constraint were monitored.

It is easy to see that this is an extemely resource-intensive approach; information has to be communicated between the two programs before and after each step, and this requires the addition of many additional steps for each step in the original program. Furthermore, communication between processes is often much slower than actual computation. Using the above analogy, it is therefore evident that monitoring all aspects of a program's behavior is a practical impossibility. (In fact, the problem can be far worse for a system that really does use semantic constraints, because there are constraints for most or all execution paths. Simply maintaining this volume of semantic information can impose excessive resource demands even if the original program was relatively small by modern standards.)

Our approach is to monitor only a subset of the possible behavioral constraints. Diversity, in the sense of Section 3, is created by selecting these constraints at random (perhaps according to a non-uniform distrbution), or, if there really are redundant copies of the program running in tandem, having different replicas monitor different constraints.

The second issue mentioned above is that we cannot obtain a tractable, abstract description of a program's behavior. If we could do so, we could easily determine which programs halt and which ones do not, and this is known to be impossible [25]. In practice, there are a number of constructs, including recursion and pointers, that are hard to handle abstractly. Our solution to this problem is, once again, that we do not monitor all aspects of a program's behavior. Our approach of only monitoring *selected* constraints allows us to simply omit constraints for programming constructs that our system cannot handle.

An additional advantage of this approach is that it makes our system incremental. If some aspects of a program's behavior can be abstracted *in prinicple*, but we have not yet implemented the necessary functions in our constraint extraction system, this does not hinder our approach. It simply works with whatever semantic information is available.

5.2 Semantic constraints

This section describes the format of the semantic constraints our system extracts from software. Currently, the system works with `C/C++` programs, but the actual constraint extraction algorithm uses an intermediate representation, a static dataflow graph, which is language independent.

In our system, a constraint is comprised of three parts

1. The antecedant, which describes a particular control path within a program. Usually, this is not a complete control path from the entry point of the program to an exit point, but rather a path from the entry point of some function to some statement within a function.

2. A consequent, which identifies a statement that will be executed if the control path described in the antecedant is taken.

3. A set of data dependancies, which describe where the variables used in the consequent statement were last defined, assuming the path described in the antecedant has been taken.

The structure of the antecedant is perhaps the most complex aspect of our constraints. An antecedant contains a number of decision nodes, which correspond to boolean-valued expressions in the original program. The only boolean expressions included are those that cannot, themselves, be decomposed into simpler boolean expressions. Each decision node is annotated with a label `true` or `false`, describing what value the corresponding expression must have for the program to follow the path that the antecedant is describing.

To allow the characterization of loops, each element of a constraint (the decision nodes, as well as the statement nodes, and the variables in the data-dependancies) is also annotated with a series of loop subscripts. The number of loop subscripts is the same as the number of loops that enclose the statement, decision, or variable being subscripted. Conceptually, each subscript describes how many times the loop in question has been executed on the given control path, meaning that these subscripts are part of the path description. However, such a subscript takes on only one of three values: 1, denoting the first iteration of the corresponding loop, n, denoting any other iteration than the first, and $n-1$, denoting the previous iteration, that is, the one that occured before the one referred to by an n subscript. In short, loop subscripts provide the information needed to describe the behavior of loops inductively: a constraint that has "1" subscripts is used to establish the base case, and constraints containing "n" and "$n-1$" loop subscripts describe the inductive step.

For example, consider the following simple program:

```
main()
{
   int a;

   for (int i = 0; i < 10; i++)
      a = a + 1;

   b = a;
}
```

The constraints pertaining to the variable `a` are given below. Each constraint has been textually expanded to make it easier to read; the section labelled `PATH` describes the antecedant, the section labeled `STATEMENT` describes the consequent, and the section labeled `DEF-REF PAIRS` describes the data dependencies.

```
PATH:
   {ENTRY: main( )} loc=0 -- ALWAYS-->
STATEMENT:
   {int a = 0 ; }loc=4
DEF-REF PAIRS:

PATH:
   {ENTRY: main( )} loc=0 -- ALWAYS-->
   {i < 10 } loc=9 loops=0  -- TRUE-->
STATEMENT:
   {a = a + 1 }loc=12 loops=0
DEF-REF PAIRS:
   Variable: a DEF: {int a = 0 ; }loc=4



PATH:
   {ENTRY: main( )} loc=0 -- ALWAYS-->
   {i < 10 } loc=9 loops=N-1  -- TRUE-->
   {i < 10 } loc=9 loops=N  -- TRUE-->
```

```
STATEMENT:
   {a = a + 1 }loc=12 loops=N
DEF-REF PAIRS:
   Variable: a DEF: {a = a + 1 }loc=12 loops=N-1

PATH:
   {ENTRY: main( )} loc=0 -- ALWAYS-->
   {i < 10 } loc=12 loops=0  -- FALSE-->
STATEMENT:
   {b = a }loc=20
DEF-REF PAIRS:
   Variable: a DEF: {int a = 0 ; }loc=4

PATH:
   {ENTRY: main( )} loc=0 -- ALWAYS-->
   {i < 10 } loc=12 loops=N-1  -- TRUE-->
   {i < 10 } loc=12 loops=N  -- FALSE-->
STATEMENT:
   {b = a }loc=20
DEF-REF PAIRS:
   Variable: a DEF: {a = a + 1 }loc=15 loops=N-1
```

There are five constraints pertaining to the variable `a`:

1. The first constraint refers to the initialization of `a` on the third line of the program. The path contains a statement saying that the entry to the function `main` is always executed and that this is the only requirement for reaching `a`'s initialization. ("Always" constraints and "never" constraints are logically somwehat superfluous, but they are useful when the constraints are manipluated by other programs. For example, this particular constraint can be used to determine which function the constraints are for.)

2. This constraint states that if the condition `(i < 10)` on line 5 is true, before entering the loop on lines 5 and six, (e.g., the zeroth iteration), then the assignment `a = a + 1` on line six will be executed for the first time. The data dependancy tells us that, prior to the execution of this assignment statement, `a` will have the value given to it at location 4 (on the third line). Note that this value can be obtained from the first constraint.

3. This constraint describes the loop inductively. If the condition `(i < 10)` was true on the previous iteration and it is still true on the current iteration, then the assignment statement on line six will be executed. In our system, the constraint `{i < 10 } loc=9 loops=N-1 -- TRUE-->` implies that location nine, associated with the condition `(i < 10)`, was actually executed on the $n-1$st iteration, though the formatting of the constraints above does not make that fact explicit. Because of this, the constraint does not refer to the first loop iteration, and the value of `a` does not come from line three. Instead it comes from the previous iteration of line six, as the data dependency states.

4. This constraint describes the assignment on line eight for the path where the loop is not executed. The preconditions of the constraint describe an infeasible path, but infeasible paths are an inevitable part of symbolic execution since their detection leads to undecidable problems. (Moreover, our constraint generation system, which created the constraints in our example, is not also tasked with the semantic analysis of those constraints.)

5. This constraint describes the assignment statement on line eight for the path on which the loop is executed. It states that, for any n where the condition on line six is executed but evaluates to `false`, and where the condition was executed on a previous iteration and evaluated to true (implying that the

loop was entered), the assigment on line six will be executed and the value of `a` in that assignment will be the one assigned on line six during the $n - 1$st iteration.

The constraints given here do not contain enough information to find the final value of `b`, but this is only because the constraints not involving `a` were omitted for brevity. The full constraint set allows a simple, brute-force deduction of `b`'s value by forward chaining. Such an approach involves several instantations of the loop constraints — the second and third constraints in the example — in which n is replaced by actual numbers. The information given is also sufficient for an approach that unravels the loop by induction, but we have not yet implemented such a system.

5.3 Constraint Extraction System

Our constraint extraction system builds semantic constraints from the static dataflow graph (SDF) of a program. A static dataflow graph encodes the dataflow of a program (as the name suggests), and thus allows us to extract the constraints described in the previous section. (The advantage of using the SDF as an intermediate representation is that it abstracts away many language dependencies).

The SDF is a graph whose nodes represent syntactic constructs in a program (usually statements or expressions), and whose edges represent the flow of control between these constructs.

The nodes isolate the information needed to determine the *semantic* behavior of the program at a particular time. The following syntactic constructs are represented as nodes in the SDF:

- expressions that represent variable definitions, variable uses, and variable declarations.
- starting points of loops.
- expressions whose value affects the flow of control.

The SDF has a directed edge from node A to node B if the code represented by node A can be executed immediately before the code represented by B, without the intervening execution of code represented by other nodes. To determine what loops enclose a statement (as is necessary for the generation of the constraints we described in the previous section), we can simply follow the control-flow paths from a statement to check if they lead back to the start of a loop.

However, the SDF does not have a representation for the flow of control in and out of functions when they are called. In addition, definitions and uses of globally declared variables are not represented. In short, the SDF does not capture all information needed to get a complete description of the program's behavior. Our system is designed to work even when limited behavioral information is available (see Section 6.1.1), so while the addition of more information can improve its performance, the absence of some information does not break the system.

6 Applications of the Mutation Scripting System

The mutation scripting system described above was intended to be a by-product of this research that would be useful in and of itself. Over the years, RST has developed a large number of software analysis systems that instrument source code, used for such diverse purposes as coverage analysis, fault-injection, and automated test data generation as well as software mutation. In the past, the software that performs source-code instrumentation has taken the form of compiled, executable code, and it proved somewhat time-consuming to modify such a system when the need arose for new types of software instrumentation. The mutation scripting system was intended as a general-purpose language that would allow new software instrumentation techniques to be implemented as *scripts*. The idea was to isolate those parts of the instrumentation technology that change from one system to the next from those that are always the same. This simplifies the task of the person designing a new software instrumentation prototype.

The mutation system has the overall aim of letting a developer specify what source-code constructs should be modified during instrumentation, and how they should be changed, not worrying about how these constructs were actually found and modified in the source code. With our existing technology, this had been somewhat difficult because the code that performs the instrumentation was woven into a traversal of a parse tree built from the source-code being instrumented.

Besides allowing tailored source-code instrumentation, such technology has a number of other potential applications:

Generalized patch scripts It is often necessary for software developers to release *patches* for existing code, either to upgrade the code or fix bugs. But existing patch systems are text based; a software patch might contain instructions like "replace the 103rd line of the program with the line `if (x >= 10)`," for example.

But suppose it were desired to replace all calls to the function `strcopy()` in a `C/C++` program with calls to `strncopy()`. (`strcopy` is a function notoriously vulnerable to buffer-overflow attacks because it copies a string of characters to a new memory location without checking if it overwrites too much memory. `strncopy()` also copies strings of characters but it does bounds-checking.) This sort of replacement is difficult (in the general case) for text-based systems because the sequence of charaters "`strcopy`" in the source code doesn't necessarily indicate an actual call to the `strcopy` function. Also, the arguments of the `strcopy` need to be changed in a context-sensitive way to make a successful call to `strncopy`.

In order to make such patches to source code automatically, a patching system needs to know about the syntactic structure of the source code being modified. Our mutation scripting system *is* aware of a program's syntactic structure and might thus be used as a basis of more powerful software patches.

Data collection and insertion In security-related applications such as intrusion detection, it is often desirable to collect data about a program while it is running. When source-code is available, it is once again appealing to instrument the source-code in order to collect this information, but such instrumentation is easier to carry out using a syntax-based software instrumentation system.

In unit-testing (testing software one function at a time), we want to observe the return values of functions (as well as other data that the function modifies), and we also want to change the *parameters* of the function, which essentially provides different test inputs to the function. Usually this is done by writing `test drivers` that isolate the component being testing while simulating the software where the component would normally be used. Source-code instrumentation — if it is easy enough to do — can allow testers to change the parameters of a component and monitor its behvior *in place*, that is, within the context of a larger program. This may be simpler than writing test harnesses, and it also helps alleviate concerns that the test harness might fail to provide an accurate picture of what would happen if the component were actually part of a larger program.

Tailored assertions and watchpoints A number of technologies developed at RST involve the use of *assertions* that signal when software enters an unsafe, insecure, or failure-prone state. This is also done with software instrumentation. But if such a tool is used by third parties who cannot recompile the source-code instrumentor, then those third parties are limited to the types of assertions that the original developers anticipated. If assertions could be placed using generalized instrumentation scripts, novel types of assertions could be automatically placed into source code without recompiling the assertion-placement tool. Thus, an assertion-placement tool that uses instrumentation scripts might be quite a bit more useful than current assertion-placement technology. Mutation scripting can provide this capability.

6.1 Virtual Redundancy: A Constraint Monitoring System

One application of semantic constraints is the monitoring of programs during execution. Many malicious attacks invlove altering a program's behavior so that it performs actions on behalf of the attacker. The most notorious of these are buffer-overflow attacks, but programs can also be attacked in some cases using heap-overflow attacks or by modifying the image of the executable in memory. In contrast, viruses modify executable programs on disc.

Constraint monitoring is the process of monitoring software behavior to verify that its behavior conforms to that specified by semantic constraints. To descibe the process succinctly, the variables that appear in the semantic constraints are monitored while the program executes. A constraint monitor verifies that the values of these variables are consistent with the behavior described in the constraints.

The purpose of constraint monitoring is to detect deviations between the actual behavior of the program and the behavior described in the source code. Since the constraints encode the semantics of program as described by the source code, subsequent modifications of that behavior — whether caused by a virus, a buffer-overflow attack, or some other malicious activity — will show up as deviations from the behavior encoded in the constraints, and this allows the constraint monitor to act as an intrusion detection system. (Other potential uses will be outlined in Section 6.1.1 below.)

Our constraint monitoring system uses the above constraints, in conjunction with source-code instrumentation of the original program, to confirm that the program is, in fact, behaving as the constraints require. An external constraint monitor, running in tandem with the original program, monitors decision points, data definitions, and data uses. This information is used to ensure that the consequents and data dependencies of any constraint are satisfied whenever the antecedent becomes true. Since the consequent and data dependencies may not become true at the same time as the antecedent, there is a timeout that delays the raising of an alarm when information is not received. An alarm is also raised if incorrect information is received from a program.

What makes constraint monitoring different from other types of intrusion detection is the *fine focus* of the monitoring activity. That is, very minute deviations from normal behavior can be detected. For example, the prototype we built under this contract detects changes such as a replacement of `x++` with `++x` in `C/C++` programs. Furthermore, a constraint-based intrusion detection system is not susceptible to false alarms, since it encodes only the behavior specified in the source code.

In a sense, using a constraint monitor is like running two copies of a program in tandem, having one program verify the actions of the other at every step. Indeed, this type of *virtual redundancy* also provides diversity, since the attacker must compromise the constraint monitoring system to remain undetected.

The question is whether this type of "virtual redundancy" really deserves the name. That is, can we achieve the other benefits of redundancy as well with a constraint monitoring system? Can we, for example, implement voting or other redunancy-based paradigms to correct abberrant behavior? Can we obtain a high degree of diversity with virtual redundancy? (We noted that having the constraint monitor run in tandem with the original program introduces some diversity, since the constraint monitor works differently from the original program, but that only gives us twofold diversity.)

We will argue below that these benefits can be obtained through virtual redundancy as well. But first, we have to address a drawback of the constraint monitoring apporach which affects the way virtual redundancy would be implemented in a practical system.

6.1.1 The practicality of constraint monitoring, and the role of diversity

As we have suggested already, this constraint monitoring system is sensitive to very small aberrations in a program's behavior. For example, as small a change as changing a `C/C++` preincrement to a postincrement triggers an alarm when the code is executed, assuming that the code in question is also being monitored. Presumably, this makes the code especially resilient to viruses, though we did not have a `Unix` virus with which to test this. The malicious attacks we tested against our system were buffer-overflow attacks, which are detected because the flow of information to the constraint monitor is interrupted.

As an anomaly detection system, the constraint monitor is much more sensitive than other program-based anomaly detection systems we know of, since they only monitor the interaction of the program with the operating system. In addition, the constraint monitor knows the exact semantics of the program, so false positives are only possible when unexpected delays cause a timeout alarm.

A practical problem with the constraint monitoring system we have described is that the number of constraints can be large. For example, the well-known toy program `triangle` generates about 600 constraints. For real programs, the number of constraints can quickly become intractable, given that the constraint monitoring system has to work in real time. This is not a shortcoming of our system in particular; it is observed whenever there is a need for abstract representations of program behavior that can reasoned about automatically. (In particular, *symbolic execution systems*, see [7, 22] for example, run into this problem.)

However, another straightforward approach to reducing the number of constraints that must be monitored — an approach consistent with the idea of redundancy and diversity — is to monitor only a randomly selected *subset* of the available constraints. This gives an attacker a chance to avoid detection, but the paradigm of diversity with randomization, described in Section 3, takes this into account. In redundant systems, different aspects of program behavior can be monitored in different replicas, but the technique can also be applied to a single program for which the constraints being monitored periodically change. With this approach, diversity could be achieved by having different versions of a program (installed at different sites, perhaps) monitor different aspects of program behavior. Attackers may be able to subvert some replicas of the monitored program, but the ultimate aim of diversity is achieved, because there is a certain *probability* that a given attack against a given system will be detected.

By monitoring only some constraints, we also overcome one of the traditional problems encountered during abstract analysis of software semantics: it is not currently possible to reason about all possible aspects of software semantics. In fact, it is one of the classical results of theoretical computer science that no such analysis is possible [23]. In practice, difficulties are encountered during the analysis of indefinite loops and pointers (with possible aliasing).

Note that it is useful if attackers can be kept in the dark about which aspects of a program's behavior are being monitored. This is a useful tool for software diversification if changes can be made quickly enough, but if it takes too much time the approach deteriorates into nothing more than security by obscurity. But it is relatively easy to change the constraints monitored by a system like ours; certainly easier than other potential approaches to diversity that involve recompiling the applications that are supposed to be diverse.

An additional benefit of such an approach is that the constraints that are successful in detecting an attack can easily be disseminated, so that even systems that were not able to detect the attack become able to do so. New replicas of a program are thus immune to past attacks.

In the next section, we will discuss how these ideas might be applied in a practical intrusion-resistant system.

6.1.2 Diversity in a single executable unit

Early in this report, we discussed the idea of diverse and redundant executables as a way to delay and potentially thwart malicious attacks. We saw that if a system remains vulnerable to old attacks, then an attacker may be relatively successful even though the system being attacked contains redundancy.

In practical terms this is a worst-case scenario. The attacker has to have the wherwithall to develop new attacks. At the current time, new attacks are developed by a relatively small core of individuals, scripted, and replayed by other attackers with less technical ability. Under this attack paradigm, the development of new attacks is quite a bit more expensive than the replay of old attacks. We are accustomed to seeing large sectors of the information infrastructure undermined very quickly by just a single, scripted attack or virus, and it does not seem as though this kind of rapid, catastrophic destruction could be carried out against a redundant system.

However, the intent of this project was to protect information systems against *stratgeic* attacks, and by that we mean attacks carried out by technically skilled individuals who are capable of developing new attacks and intent on doing so until some target system has been successfully subverted.

To protect against such attacks, it would be desirable for an information system to recover from past attacks, and to do so in such a way that after recovery, the system is no longer vulnerable to the same attacks. In other words (according to the results described earlier in this report), each individual *replica* in a redundant system should be able to reconfigure itself after an attack.

This paradigm of dynamic reconfiguration has an additional advantage: systems could be reconfigured even when *no* attacks have taken place, in order to impair the attacker's intelligence-gathering efforts. Indeed, red-team experiments carried out by DARPA have suggested that this approach is effective. The easier it is to reconfigure a system, the more often reconfiguration can take place, and the more effective this approach becomes.

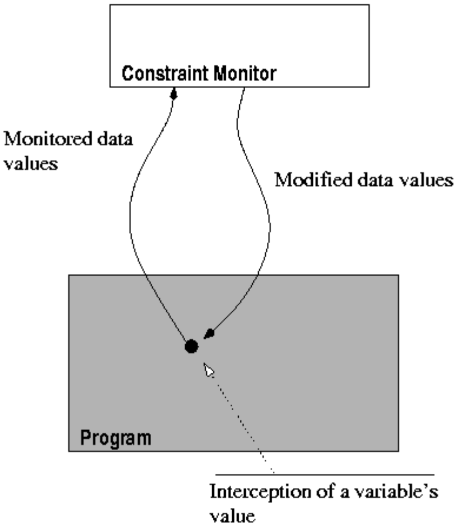

Figure 4: Illustration of a constraint monitor being used to intercept, monitor, and possibly alter the value of a program variable. The program has been instrumented to give the constraint monitor access to the variable value, and when the program's execution reaches the point where the variable is referenced, execution temporarily passes to the constraint monitor, where the value is checked and possibly modified before control is returned to the original application. For simplicity, the figure only represents the interception of a single variable's value at a single location.

In the DARPA red-team experiments, the apparent network topology of a system was dynamically reconfigured at various intervals to delay the progress of attacks. This was prohibitively difficult to implement. But this proposal deals with redundancy and diveristy of executable applications, and it might be quite a bit harder to implement a system that quickly and automatically mutated the applications on a single information system.

This leads to the idea of introducing diversity into a single executable program. Here, the application itself does not change, but certain aspects of its behavior do change at random. The changes may effect the behavior of the application itself, or it may just affect the measures used to detect and/or recover from attacks. Semantic constraints would be a key element of this approach.

The idea is illustrated in Figure 4. The original application is essentially unchanged, but it is augmented with a second system that monitors and potentially alters its behavior. This second system has built-in diversity because its behavior is reconfigurable. Below, we will discuss exactly what functions this second susbsystem might perform in order to provide the benefits of diversity, but we start by noting the two inherent benefits of this approach:

1. Two copies of the same application, running on different hosts, are slightly different in their behavior. This makes it harder for a single attack to be repeated against both hosts.

2. A given copy of the application changes its behavior over time. This makes it harder to repeat the same attack twice against the same host.

.

To understand this approach, it is necessary to understand the details of the constraint monitoring subsystem in more detail. Recall that the monitoring subsystem looks at the application's internal state to ensure that the application is behaving as it should be. This monitoring is accomplished by *instrumentation* in which small fragments of the program are essentially *wrapped.*

For example, if a program contains the statement

```
x = y + 1;
```

and the goal is to monitor the value of the expression y+1 when that statement is executed. Monitoring is accomplished by instrumenting the code as follows:

```
x = wrap_integer_expression(y + 1);
```

The function wrap_integer_expression is used to record the value of y + 1. Normally, wrap_integer_expression returns the value of y + 1 to ensure that the behavior monitor does not interfere with the normal operation of the program.

However, wrap_integer_expression could also return a *different* value. Therefore, if many or all of the expressions in a program are wrapped in this way, we can exercise very fine-grained control over the program's behavior. Thus, the behavior monitoring system allows diversity to be introduced in a number of ways.

Detection of anomalous behavior The behavior monitoring system can serve a a fine-grained intrusion detector, as was already discussed above. The idea is to ensure that the program's behavior conforms to the behavior expressed in the semantic constraints extracted from the program. Normally, the program should do exactly what the constraints say since the constraints were obtained from the program itself. But many attacks involve some type of *modification* by the attacker of the program's behavior. Stack-smashing attacks [21] are a notorious example. Even subtle changes in program behavior can be detected as deviations from the behavior described in the constraints.

We have already pointed out that it is impractical to monitor *all* aspects of program behavior, but dynamic reconfiguration of the monitoring subsystem makes it difficult to anticipate which aspects of its behavior are being monitored.

Detection of modified code Many attacks work by modifying existing programs. For example, when a virus infects an executable file, it does so by adding its own malicious functionality to the application. Some attacks also involve modifying the binary image of system applications while they are cached in memory accessible to a malicious user. For a constraint monitoring system, such modifications show up as runtime behavior changes, and can be detected as in the previous case.

Recovery from attacks In the above examples, attacks are detected but nothing is done about them. They are essentially just intrusion detection systems that rely on some other mechanism to respond to the intrusion.

However, a constraint monitoring system knows how a program *should* behave and it can correct that behavior by reinjecting the correct data-state. This appears feasible because (a) each constraint deals with only a limited number of different aspects of the data-state, and (b), those parts of the data-state are being monitored already, and past experience suggests that when software instrumentation is used to perform such monitoring, it is often straightforward to also *modify* the values that are being monitored.

7 Conclusion

To discuss the practical implications of our analytical results, it is useful to (informally) define two threat models.

- The attacker exploits a vulnerability that is visible in some artifact that is (or is supposed to be) subjected to validation during the software development process.
- The attacker exploits a vulnerability that is not visible in any such artifact.

In the first case, the vulnerability might be categorized as a sort of software flaw; it is analogous to any traditional software flaw except that it leads to the violation of a security-oriented specification rather than the violation of a correctness-oriented specification. Although assuring software correctness is difficult enough, there is hope (at least) that this type of threat can be dealt with using traditional methods for software assurance (with the exception of statistically-based techniques, since these inherently assume a non-malicious adversary).

In the second case, which unfortunately seems to be the predominant one in practice, the attacker exploits a vulnerability resulting from behavior not visible in the software artifacts being examined. For example, high-level representations of software (such as the software source code) abstract away the details of the parameter passing mechanism, but buffer-overflow attacks [21] exploit that mechanism. Today we know that certain high-level programming constructs lead to buffer-overflow vulnerabilities, so in some sense these vulnerabilities can now be classified as software flaws. The greatest difficulty seems to be caused by vulnerabilities that are both unknown and invisible in any software artifacts that are subjected to inspection.

In principle, automated diversity seems to be a good way to address such vulnerabilities. To be sure, the diverse software has to adhere to certain requirements and specifications, since the replicas will not be interchangeable otherwise. If vulnerabilities exist in these specifications they will simply be replicated across all copies of the software. But such vulnerabilities can often be detected by other methods: to automate the diversification process, we need some sort of written description of what a software module does, and that written specification can be scrutinized. To address the second (and more dangerous) class of vulnerabilities listed at the start of this section, we need to alter those aspects of program behavior that are *not* outlined in any specification, and that is precisely what automated diversification does.

The drawback of the technique employed in this project was that mutations needed to be specified explicitly. This is undesirable because very many things need to be mutated in order to get a useful increase in survivability. In a sense — going back to the two classes of vulnerabilities described at the start of the section — the person who specifies the mutations must think of everything that was overlooked by the person inspecting the code. The difficulty of doing this is the very thing that makes certain vulnerabilities so dangerous in the first place.

One way to solve this problem is to redesign the entire system from the ground up, resulting in what we have characterized as *hyperdiversity*. But the reasoning above seems to imply that survivability can be improved *only* in this way. With current technology, such a solution cannot be automated.

In summary, it seems that automated diversification seems to require an *intensional* mutation approach — one where it is only necessary to provide implicit descriptions of the mutations. An example of intensional mutation might be a directive that says “replace all interprocess communication mechanisms.” In contrast, current software mutation technology is *extensional* in the sense that each mutation has to be described explicitly. We can automatically modify *certain aspects* of all IPC mechanisms, but to truly “replace all interprocess communication mechanisms” more than once, a human has to implement several interprocess communication mechanisms by hand because each primitive has to be implemented in a different way. As far as we know, this cannot currently be automated.

Before concluding, we should point out that there are other forms of diversification that are not encompassed in the summary above and appear, at least on the surface, to be more readily implementable. For example, *temporal diversity* might be used to prevent simultaneous execution of the same data on different systems. In some sense computer networks already possess temporal diversity, and indeed one might argue that many past worm and virus attacks would have been far more catastrophic if it were not so. But truly effective temporal diversity would require a response mechanism that protects those systems that are not immediately compromised. *Data diversity* [2] is another potential solution; this essentially means mutating data instead of code while, once again, preserving the original meaning.

References

[1] A. T. Acree. *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

[2] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. In *Proc. FTCS '87*, pages 122–126, 1987.

[3] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical Report 276, Department of Computer Science, Yale University Research Report, 1979.

[4] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Learnability and the vapnik-chervonenkis dimension. *JACM*, 35(4):965–984, 1989.

[5] T. A. Budd. *Mutation Analysis of Program Test Data.* Phd Thesis, Yale University, New Haven, CT, 1980.

[6] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, 5(4), July 1979.

[7] L. A. Clarke. A system to generate test data symbolically and execute programs. *IEEE TSE*, SE-2(3):215–222, Sept. 1976.

[8] Lori A Clarke and Debra J Richardson. Symbolic evaluation methods for program analysis. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 9, pages 264–300. Prentice-Hall, 1981.

[9] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[11] P. D'haeseleer, S. Forrest, and P. Helman. An immunological approach to change detection: Algorithms, analysis and implications. In *IEEE Symposium on Security and Privacy*, 1996.

[12] A.K. Ghosh, A. Schwartzbard, and M. Schatz. Using program behavior profiles for intrusion detection. In *Proceedings of the SANS Intrusion Detection Workshop*, February 1999. To appear.

[13] Nevin Heintze. Set constraints in program analysis. Technical report, Carnegie-Mellon University, July 1993.

[14] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[15] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, April 1997.

[16] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE TSE*, SE-12(1):96–109, January 1986.

[17] M. R. Lyu, editor. *Handbook of Software Reliability Engineering.* McGraw-Hill, New York, 1995.

[18] L. J. Morell. Theoretical insights into fault-based testing. In *Proceedings, 2nd Workshop on Software Testing, Verification, and Analysis*, pages 45–62, 1988.

[19] J. D. Musa. Operational profiles in software engineering. *IEEE Software*, 10(2):14–332, 1993.

[20] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *Software — Practice and Experience*, 29(2):167–193, February 1999.

[21] Aleph One. Smashing the stack for fun and profit. Online. Phrack Online. Volume 7, Issue 49, File 14 of 16. Available: `www.phrack.com/Archive/`, November 9 1996.

[22] C. V. Ramamoorty, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE TSE*, SE-2(4):293–300, Dec. 1976.

[23] H. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 89:25–59, 1953.

[24] E. H. Spafford. The internet worm program: An analysis. *Computer Communications Review*, 19(1):17–57, April 1989.

[25] Alan M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc. (2)*, 42:230–265, 1935.

[26] V. Vapnik and A. Y. Cheorvonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, XVI(2):264–280, 1971.

[27] J. Voas, F. Charron, G. McGraw, and K. Miller M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, 14(4):73–83, July 1997.

[28] J. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley and Sons, New York, 1998.